



Copyright © Hal Pomeranz

This material is distributed under the terms of the
Creative Commons Attribution-ShareAlike 4.0 International License
<http://creativecommons.org/licenses/by-sa/4.0/>

Download updates from <https://archive.org/details/HalLinuxForensics>

Hal Pomeranz
hrpomeranz@gmail.com
@hal_pomeranz@infosec.exchange

v3.0.2



Attribution-ShareAlike
CC BY-SA

WHO IS HAL POMERANZ?

Started as a Unix Sys Admin in the 1980s
Independent consultant since 1997
Digital forensics, incident response, expert witness
Have done some interesting Linux/Unix investigations

hrpomeranz@gmail.com
@hal_pomeranz@infosec.exchange



Attribution-ShareAlike
CC BY-SA

From the "official bio":

Hal Pomeranz is an independent digital forensic investigator who has consulted on cases ranging from intellectual property theft, to employee sabotage, to organized cybercrime and malicious software infrastructures. He has worked with law enforcement agencies in the United States and Europe, and with global corporations.

While perfectly at home in the Windows and Mac forensics world, Hal is a recognized expert in the analysis of Linux and Unix systems and has made key contributions. His EXT3 file recovery tools are used by investigators worldwide. His research on EXT4 file system forensics provided a basis for the development of Open Source forensic support for this file system. Hal has also contributed a popular tool for automating Linux memory acquisition and analysis. But Hal is fundamentally a practitioner, and that's what drives his research. His EXT3 file recovery tools were the direct result of an investigation, recovering data that led to multiple indictments and successful prosecutions.

Raised in the Open Source tradition, Hal shares his most productive tools and techniques with the community via his GitHub and blogging activity.



LINUX IS EVERYWHERE

- Cloud instances
- Embedded devices ("IoT")
- Android
- ChromeOS

Whether they realize it or not, people interact with Linux systems every day. The Internet runs on Linux, whether it's core DNS services, popular web sites, and file and video sharing. The embedded devices in their homes—DVRs, network equipment, smart appliances—are often running Linux. And Android devices make Linux the dominant OS platform in terms of installed devices by a huge margin.

Because the owners of the Linux devices may not fully understand the operating system and how to secure it, many of these devices are easily compromised. As this equipment becomes more powerful and more connected, it presents an opportunity for attackers. We have already seen powerful botnets like Mirai, and extensive cryptocurrency mining operations running on compromised Linux systems. Ransomware is increasingly targeting Linux infrastructures.

This course introduces Linux system forensics, with a primary focus on Linux servers. We will cover both "live" and "dead box" analysis, and key Linux artifacts that are useful in many sorts of investigations. The course uses Open Source forensic tools, but the investigative techniques are applicable to any forensic tool chain.

WHAT'S DIFFERENT ABOUT LINUX?

No registry

Have to gather system info from scattered sources

Different file system

Important metadata zeroed when files deleted

Access time updates are intermittent

Older file systems lack file creation dates

Files/data are mostly plain text

Good for string searching & interpreting data

While Windows tends to concentrate configuration information in the registry, things are much less centralized in Linux. Every application and Linux subsystem tends to have separate configuration files and installation directories. Part of Linux forensics is knowing where the most important artifacts are.

Linux has its own file systems. EXT4 is most common, but Red Hat is now using XFS as its default file system. ZFS is another option with a decent installed base. While the EXT family of file systems tends to have decent forensic tool support, support for XFS and ZFS is much less available. Linux file systems have different timestamp rules from Windows NTFS, and recovery of deleted data is more challenging because Linux file systems zero out file metadata upon deletion.

On the plus side, most of the data in Linux is plain ASCII text. Searching for and correlating data tends to be easier than in other OSes.



LIVE CAPTURE WITH UAC

Sometimes live capture is the right way to go. UAC is an excellent, easily extensible tool for performing live captures from Linux and Unix-like operating systems.



THE CASE FOR LIVE TRIAGE

Acquiring full disk images may not be practical

- Images too large

- Bandwidth too limited

- Too many systems to investigate

Memory forensics can be tricky

- Kernel version issues

- Lack of dependencies

- Organizational/process issues

Go "old school" and try live collection!

In the 1990s capturing the live state of a system generally involved running scripted commands on the machine to capture process information, network connections, etc. The rise of efficient memory forensic tools like Volatility™ created a movement away from live collection to a more memory-based approach.

However, there are some limitations to memory forensics on Linux, particularly around the creation of appropriate memory analysis profiles. Also, acquisition of memory images can be difficult in some environments, particularly as we see system memory sizes approaching 1TB and beyond. We will discuss these issues in more detail in later sections of the course.

To do investigations at enterprise scale, we need a way to quickly capture the most important artifacts in a way that can be scripted to run rapidly across a hundreds or thousands of systems. This leads us back to the "old school" method of running targeted commands on live systems to collect crucial pieces of evidence that can be used to confirm compromise and/or investigate the system without the costs of doing complete memory or disk acquisition.

PROS –

- Capture critical information in just a few minutes
- Capture live system state, including processes, network connections, etc
- Happens on live system without interrupting normal jobs
- Typical data capture size is a few GB or less
- Can be done over low-bandwidth and/or high-latency links
- Can be automated

CONS –

- Will change the state of the machine by running jobs and saving data
- Rootkits may interfere with data being captured
- May need tuning based on local site's admin practices
- Not a complete image (e.g., no unallocated, etc)

DON'T REINVENT THE WHEEL

UAC – live collection tool for Unix-like operating systems

Self-contained:

- Copy tool archive to target host
- Uncompress archive on target
- Run tool as root on target
- Collect output archive file from target

Configurable via profile file and/or command-line

UAC is my current favorite live capture utility. It's easily extensible and configurable. It supports other Unix and Unix-like operating systems besides Linux (MacOS, *BSD, Solaris, AIX, e al). The primary maintainer is Thiago Canozzo Lahr, who has been very responsive to feature requests and suggestions. UAC is hosted at <https://github.com/tclahr/uac>

Of course, there are other live collection tools available. A good summary of available tools is located at:

<https://github.com/swisscom/ArtifactCollectionMatrix#linux-live-collection-tools>

UAC is distributed in a *.tar.gz file. No compilation is needed. Simply copy the compressed tarfile to the target machine and unpack it wherever you choose. Run the UAC tool as root, specifying a profile file that lists the artifacts you wish to collect. The specified collections are performed and the results are put into another *.tar.gz file that you can easily copy to another system for analysis.

On a small virtual machine, a "full" collection (without memory) ran in under 10 minutes. The resulting *.tar.gz file was approximately 20MB. Obviously, collection times and size of data collected will vary depending on the target system.

RUNNING UAC

```
# ./uac -a memory_dump/avml.yaml -p ir_triage /root
```

ir_triage

Good starting point for system collection

full

Adds user browser and application artifacts

UAC comes with two basic pre-defined profiles:

- "ir_triage" – captures important system artifacts needed for typical investigations
- "full" – everything in "ir_triage" plus browser and other user application artifacts

Use the "-p" option to select the profile you prefer.

In addition to the items specified by the profile, you can also capture additional artifacts using "-a" to specify what you want to collect. For example, UAC includes the AVML utility for capturing RAM and a YAML configuration file for running it appropriately.

Profile files are stored in the "profiles" subdirectory of the UAC distribution. They are text files in a simple YAML format and are easily modified to suit your needs. For example, if you always wanted to acquire memory you could modify the "ir_triage" profile to include the "memory_dump/avml.yaml" file instead of having to use "-a" on the command line all the time. The profile files reference the YAML configuration files in the "artifacts" subdirectory which describe how to collect various artifacts. The artifacts files are also simple text files and relatively easy to configure and extend.

For more information, see the UAC documentation at:
<https://tclahr.github.io/uac-docs/>

GETTING RESULTS

```
# ls -lh uac-LAB-linux-*
-rw-r--r--. 1 root root 482 Jul  1 21:20 uac-LAB-linux-20220701212047.log
-rw-r--r--. 1 root root 20M Jul  1 21:20 uac-LAB-linux-20220701212047.tar.gz
# mkdir uac-LAB-linux-output
# cd uac-LAB-linux-output/
# tar xzf ../uac-LAB-linux-20220701212047.tar.gz
# ls
[root]      chkrootkit      live_response  uac.log.stderr
bodyfile   hash_executables  uac.log
```

Results are saved to the target directory you specify on the UAC command line. The results file name is "uac-<hostname>-<os>-<YYYYMMDDhhmmss>.tar.gz". So you can run UAC multiple times on the same system without conflict, and you can capture multiple systems' *.tar.gz files to one big capture directory.

Note that UAC does have command-line options for automatically SFTPing the results file to a remote system, but you have to provide explicit credentials on the command line to do so. This doesn't seem like the most secure option to me. Similarly, there are options for writing the output to an Amazon S3 bucket or an Azure blob store.

When you are unpacking the results file, be sure to do so in an empty directory. Otherwise the contents of the collection will get mixed in with the other contents of your current working directory.

We'll go over the contents of the collected files in upcoming sections, but I did want to mention the "uac.log" file. This file includes timestamped entries for each acquisition job that runs. If acquisitions are taking too long, you can use these timestamps to figure out where the slow steps are happening:

```
# grep INFO uac.log
2022-07-01 21:11:32 +0000 INFO UAC (Unix-like Artifacts Collector) 2.2.0
2022-07-01 21:11:32 +0000 INFO Command line: ./uac -p full /root
2022-07-01 21:11:32 +0000 INFO Operating system: linux
2022-07-01 21:11:32 +0000 INFO System architecture: x86_64
2022-07-01 21:11:32 +0000 INFO Hostname: LAB
    [... snip ...]
2022-07-01 21:12:35 +0000 INFO Parsing artifacts file 'live_response/system/w...
2022-07-01 21:12:35 +0000 INFO Parsing artifacts file 'bodyfile/bodyfile.yaml'
2022-07-01 21:15:32 +0000 INFO Parsing artifacts file 'live_response/hardware/...
2022-07-01 21:15:32 +0000 INFO Parsing artifacts file 'live_response/hardware/...
```

The "bodyfile/bodyfile.yaml" step started at 21:12:35, and the next step didn't run until almost three minutes later. In this fashion you can find the steps that are taking the longest to run and possibly modify your profile file to leave them out.

Note that in this case I find bodyfiles to be useful artifacts to collect and I wouldn't want to skip this step. More on bodyfiles when we talk about *Timeline Analysis* later in the course.

LAB – RUNNING UAC

You know you want to!

Get some experience configuring and running UAC.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.

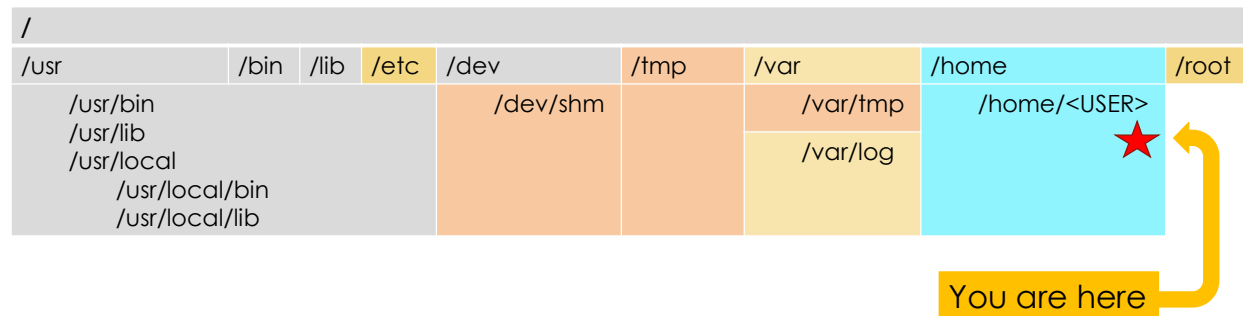


ANALYSIS: FILE SYSTEMS

Now that we've collected some UAC data, how can we use it to quickly triage a system and locate signs of a compromise? We'll start with a quick overview of the standard Linux file system and then discuss some analytic tricks for finding things that "don't belong".

Special shout-out to Tyler Hudak (@SecShoggoth) for providing UAC data from one of his honeypots. We'll be using this data for the lab exercises in these analysis sections. Thanks, Tyler!

LINUX DIRECTORIES



The majority of the Linux operating system is installed under /usr, with /bin and /lib normally just being links to /usr/bin and /usr/lib, respectively. Under normal operations, you can think of /usr as being read-only: unless the system administrator is actively updating or patching the system, the files under /usr should not change.

/etc is the primary configuration directory for the system. Nearly every service on the system has its own configuration files and/or directories under /etc. Typically you need administrative access to modify the files under /etc. /etc is also where you will find configuration files related to user accounts and privileges.

The operating system needs a place to write data that gets created while the system is running— that's the job of the /var directory. For example, system log files usually end up in /var/log. There are other directories under /var for specific system processes.

Users have their own personal directories, usually found under /home. The exception is the administrative user—the "root" user on Linux— whose home directory is /root. Each directory is usually only accessible by the specific user owner of that directory.

Finally there are temporary directories where anybody on the system may write. Many exploits that are unable to achieve administrative privileges will stage files in these "world-writable" temporary directories. The classic temp directory paths are /tmp and /var/tmp.

/dev/shm is a world-writable directory that only exists in memory— when the system is shut off or rebooted, the content of /dev/shm is lost. This makes /dev/shm a convenient place for attackers to stage files that they don't wish analysts to easily recover. It's worth checking the contents of /dev/shm before you turn off that system!

On many Linux systems, /run is an in-memory file system for storing data that does not need to persist across reboots. So far, we haven't seen a lot of attacker activity targeting /run which is a little surprising to me. Seems like a good directory to hide in. However, the attacker has to achieve elevated privileges before writing into this directory.

ANALYTIC NOTES

Executables normally live under **/usr**

/usr/bin, /usr/sbin, /usr/libexec, etc

Exceptions include 3rd-party software under **/opt**

Be suspicious of programs in user-writable directories

/tmp, /var/tmp, /dev/shm

User profile dirs under **/home**

Standard Linux programs are installed in specific directories under **/usr**— for example **/usr/bin, /usr/sbin, /usr/libexec**, and so on. Third-party software is often installed under **/usr/local**, and so you see directories like **/usr/local/bin, /usr/local/sbin**, etc. In some cases, third-party software gets installed under **/opt** (e.g., **/opt/SAP/bin**, etc).

When you see programs installed and running from other paths, you should be suspicious. This is particularly true of programs running from world-writable directories like **/tmp, /var/tmp**, and **/dev/shm**. While there are occasionally legitimate use cases for users running their own programs from their **/home/<user>** directory, there is almost never a legitimate reason to see things running from **/tmp**. Typically, programs running from **/tmp** are exploits that didn't achieve administrative privileges.

As an analogy to Windows forensics, we generally expect to see Windows programs running from **c:\Windows\System32** or **c:\Program Files***. If you see something running from **c:\ProgramData** or a user's local temp directory, you are naturally suspicious of that process.

HIDDEN FILES/DIRECTORIES

File and directory names starting with "." are hidden

Use "**ls -a**" or "**ls -A**" to see them

Directory names starting with "." are uncommon

Attackers may stage tools here

There is a Unix convention that any file or directory name that begins with a period (".") is treated as a "hidden" file and not shown in the output of the "ls" command by default. You can use "ls -a" or "ls -A" to see the hidden files and directories:

```
$ ls
$ ls -a
.  ..  .bash_logout  .bash_profile  .bashrc  .mozilla
$ ls -A
.bash_logout  .bash_profile  .bashrc  .mozilla
```

You'll note that the difference between the two commands is that "ls -a" shows the "." (current directory) and ".." (parent directory) links that are present in every Unix directory, which "ls -A" doesn't show these links.

Attackers will sometimes try making hidden directories to stage their tools. However, outside of user home directories, hidden directories are not that common. We can use a little bit of command line kung fu to spot these directories:

```
# find / -path /root -prune -o -path /home/\* -prune \  
    -o -type d -name .* -print  
/dev/shm/.rk  
/dev/shm/..  
/dev/shm/.. /.install  
    [... snip ...]  
/tmp/.ICE-unix  
/tmp/.ICEd-unix
```

The path names under /dev/shm are definitely unexpected. Note the use of a directory named ".. " (dot-dot-space) to try and blend in next to the normal ".." link present in each directory.

It turns out that /tmp/.ICE-unix is a legitimate system directory, although you'd probably have to be familiar with this directory from other Linux systems to know that. However, another team of attackers has created /tmp/.ICEd-unix to try and blend in with the landscape.

DELETED FILES AND EXECUTABLES

Attacker installs malware
Attacker executes malware
Attacker deletes executable

Malware continues running
Executable no longer appears in file system

Linux allows running an executable and then deleting that executable from the file system. You will no longer be able to find the executable by searching the file system. However, the blocks used by the executable will not be freed until the last process running that executable has exited.

Attackers will often use this trick to make it harder to find their malware. Note that if you are on the live system with malware running from a deleted binary, there is a quick method for recovering the deleted executable:

```
$ cp /bin/bash /tmp/bash
$ exec /tmp/bash
$ rm /tmp/bash
$ ls -l /proc/$$/exe
lrwxrwxrwx. 1 lab lab 0 Jul  2 20:21 /proc/25479/exe -> /tmp/bash
(deleted)
$ cp /proc/$$/exe /tmp/recovered-bash
$ md5sum /tmp/recovered-bash /bin/bash
f926bedd777fa0f4f71dd2d28155862a  /tmp/recovered-bash
f926bedd777fa0f4f71dd2d28155862a  /bin/bash
```

My bash shell continues to run even though the binary has been deleted. If you look more closely at the process directory under /proc, you can see the original executable pathname and the fact that the binary is deleted. You can use this link to recover the deleted executable too!

UAC DATA CAN HELP!

live_response/process/running_processes_full_paths.txt

Look for EXE paths not under **/usr**

Look for **"(deleted)"** binaries

live_response/process/ls_-l_proc_pid_cwd.txt

Was the attacker in a strange dir when they ran that tool?

Putting these analytic ideas together, we can begin searching for evil in the data collected by UAC.

UAC collects the `/proc/<pid>/exe` link information in the file

`live_response/process/running_processes_full_paths.txt`. With a little command line kung fu, we can look for programs outside of `/usr`:

```
# grep -F '> /' live_response/process/running_processes_full_paths.txt |  
    grep -F -v /usr  
lrwxrwxrwx 1 mail mail 0 Apr 1 17:26 /proc/1811/exe ->  
    /dev/shm/.rk/lsof (deleted)  
lrwxrwxrwx 1 mail mail 0 Apr 1 17:26 /proc/1817/exe ->  
    /dev/shm/.rk/xterm (deleted)
```

Let's see, executables installed under `/dev/shm/.rk`, executed, and then deleted. These are clearly not normal processes!

Or how about looking for processes whose current working directory is a hidden directory?

```
$ grep -F /. live_response/process/ls_-l_proc_pid_cwd.txt  
lrwxrwxrwx 1 mail mail 0 Apr 1 17:26 /proc/1811/cwd -> /dev/shm/.rk  
lrwxrwxrwx 1 mail mail 0 Apr 1 17:26 /proc/1817/cwd -> /dev/shm/.rk
```

Oh look! The same suspicious processes again!

LOOKING FOR SLEEPERS

hash_executables/list_of_executable_files.txt

Look for hidden files and directories
Look for executables outside of **/usr**

[root]

Look for hidden directories

live_response/process/lsof_-nP1.txt

bodyfile/bodyfile.txt

More detail about hidden directory content

`hash_executables/list_of_executable_files.txt` is a list of every file in the file system with an execute bit set. Note that not all of these files are necessarily actual executable programs or scripts. Sometimes files accidentally get execute permissions set on them.

```
$ grep -F /. hash_executables/list_of_executable_files.txt  
/tmp/.ICEd-unix/.src.sh
```

Here we're looking for directories and file names starting with `."`. Hmmm, `/tmp/.ICEd-unix/.src.sh` sure looks interesting!

UAC copies some critical files and directories from the target machine and puts them under `"root]"` in their standard directory layout. How about using our trick to look for directory names starting with `."`:

```
$ find \[root\]/ -type d -name .*
[root]/etc/sv/ssh/.meta
[root]/home/lab/.local
[root]/root/.cargo
[root]/root/.cargo/registry/index/github.com-1ecc6299db9ec823/.cache
[root]/tmp/.ICEd-unix
[root]/var/lib/lightdm/.local
```

There are some legitimate hidden directories under the home directory of the "lab" and "root" users, but then our /tmp/.ICEd-unix directory pops up again.

bodyfile/bodyfile.txt and live_response/process/lsof_-nP1.txt contain a wealth of detailed information about files and directories on the system. Once you have located some specific evil pathnames, you can check these files and get more information about what files exist under those directories.

```
$ grep -F /tmp/.ICEd-unix bodyfile/bodyfile.txt
0|/tmp/.ICEd-unix/.src.sh|1624563|-rwxr-xr-x|0|0|184|1680368291|16803...
0|/tmp/.ICEd-unix|1624556|drwxr-xr-x|0|0|4096|1680370054|1680368383|1680...
$ grep -F /dev/shm/. live_response/process/lsof_-nP1.txt
lsof      1811      8    cwd  DIR    ... /dev/shm/.rk
lsof      1811      8    txt  REG    ... /dev/shm/.rk/lsof (deleted)
xterm     1817      8    cwd  DIR    ... /dev/shm/.rk
xterm     1817      8    txt  REG    ... /dev/shm/.rk/xterm (deleted)
xterm     1817      8    0r   FIFO   ... /dev/shm/.rk/data (deleted)
tail      1818      8    1w   FIFO   ... /dev/shm/.rk/data (deleted)
```

Looks like the only file under /tmp/.ICEd-unix is .src.sh". The lsof data shows us the executable paths and current working directory data under /dev/shm/.rk that we saw before. But it also alerts us to the fact that there is a deleted FIFO here called "data" and another process "tail" that is interacting with it.

LAB – HONEYPOT PART 1

So much evil!

Investigate file system clues in an actual honeypot compromise. My thanks to Tyler Hudak (@SecShoggoth) for providing the UAC data.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

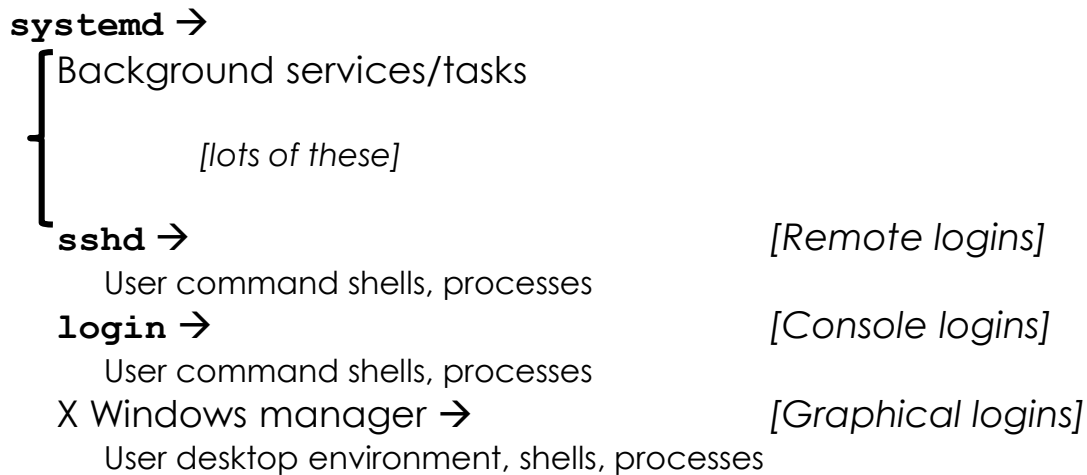
Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



ANALYSIS: PROCESSES

Understanding how Linux processes normally behave can help you spot potential problems.

LINUX PROCESS HIERARCHY



Unlike Windows, the Linux process hierarchy is generally very "flat". This is actually a good thing, since it makes trouble spots more obvious.

The modern Linux process hierarchy starts with `systemd` at PID 1. `systemd` starts every other service on the machine, including dozens of background services like web servers, DNS servers, DHCP clients, and so on. For the most part, these background services just keep running and spawn few, if any, child processes.

The most common login service on Linux is SSH. Coming out from the `sshd` process is where you commonly see Linux command shells like `bash` and user interactive commands:

```
| -sshd--+-sshd---sshd---bash
|   `--sshd---sshd---bash---sudo---uac---uac---pstree
```

Here I've got a couple of SSH sessions running, including one where UAC is running via Sudo and capturing the `ps tree` output.

In some cases, users will be logging in on the console of the machine. If they are using a text-style console session, then the user's session will come out from the "login" process:

```
| -login--bash
```

Logins via the graphical or windowing interface show very complex process hierarchies coming out from a window manager like `gdm`:

```
| -gdm-+-X---5*[{X}]  
| | -gdm-session-wor-+-gnome-session-b-+-gnome-shell-+-ibus-daemon-+...  
| | | | | | -ibus-engine-sim---2*[{ibus-engine-...  
| | | | | `2*[{ibus-daemon}]  
| | | | `20*[{gnome-shell}]  
| | | | -gsd-a11y-settin---3*[{gsd-a11y-settin}]  
| | | | -gsd-clipboard---2*[{gsd-clipboard}]  
| | | | -gsd-color---3*[{gsd-color}]  
| | | | -gsd-datetime---2*[{gsd-datetime}]  
| | | | -gsd-housekeepin---2*[{gsd-housekeepin}]  
  
[... snip ...]
```

ANALYTIC NOTES

Background services should not spawn shells!

Also look for user processes trying to masquerade

```
live_response/process/pstree.txt
```

What you should definitely NOT be seeing is user shells and interactive commands outside of these normal login contexts. If your web server is spawning an interactive shell, you're in a lot of trouble:

```
| -httpd---4*[httpd---26*[{httpd}]]  
|                               '-bash---sudo---su---bash
```

Fortunately, the relatively "flat" nature of the Linux process hierarchy makes these events stand out.

Unix-like operating systems allow processes to change their displayed name on the fly. So a process might run from the binary "MyEvilProgram" but then change its process name so that the "ps" command displays it as "[kworker/3:3]", which blends in well with other similarly named system tasks.

However, if the attacker is running this process interactively, then it becomes fairly obvious that something strange is happening:

```
| -sshd+-sshd---sshd---bash  
|   `--sshd---sshd---bash---sudo---[kworker/3:3]
```

Normally kernel threads like the "kworker" processes are not even shown in the output of "pstree". Seeing one being started by an interactive user is definitely cause for suspicion.

ORPHAN PROCESSES

Execute program as background task

Exit parent shell

Child process Parent PID (PPID) is now **systemd**

*Can show up as legit user processes running under **systemd***

Could also be malicious

The process hierarchy shows which process is the parent of each new process. But sometimes a parent process will exit but leave its children running:

```
# runbackups >backup.log 2>backup.err &
[1] 67799
# ps -ef | grep 67799
root      67799   72822   0 12:43 pts/2      00:00:00 runbackups
# exit
$ ps -ef | grep 67799
root      67799       1   0 12:43 pts/2      00:00:00 runbackups
```

Our backup job starts running as PID 67799 and its parent PID (PPID) is 72822, the PID of the root shell we ran the job from. But then we exit that root shell, and our backup job becomes orphaned. Orphan processes show their PPID as 1, which is systemd.

But this can get confusing when looking at the process hierarchy, because now our backup job appears to have been started by systemd, just like the other background services:

```
| -runbackups
| -smartd
| -sshd+-sshd---sshd---bash
|   `--sshd---sshd---bash---sudo---bash
```

This makes analysis a little more confusing. When you see an interactive program in the process hierarchy immediately under the systemd process, is this a legit user process that has simply been orphaned? Or is it a persistent job started by an attacker trying to blend in?

You'll have to use other clues about the process to determine whether it's suspicious or not. For example, the suspicious "lsof" and "xterm" processes we saw running from /dev/shm/.rk show up as orphaned processes in the pstree output we captured using UAC:

```
systemd+-...
  [... snip ...]
  |-lsof
  [... snip ...]
  `--xterm
```

SCHEDULED TASKS

Popular malware persistence mechanism

Look for tasks executing out of non-standard directories

```
[root]/etc/*cron*  
[root]/var/spool/*cron*
```

Just like we find on Windows, Linux scheduled tasks are a popular mechanism for attackers to start their malware and ensure that it stays running persistently. However, on Linux there are multiple task scheduling mechanisms, which means numerous configuration files you must check for potentially malicious jobs.

Sometimes the exploits are straightforward and easy to spot. For example, this entry in `/var/spool/cron/crontabs/root` is likely to raise suspicion because of the directory path and program name:

```
*/5 * * * * /tmp/.ICEd-unix/.src.sh
```

The first five fields specify the minute, hour, day of the month, month, and day of the week that the job should run. Here our job will run every five minutes, every hour of every day.

The problem is that things need not be so straightforward. For example, a legitimate task created by the system administrator might invoke a shell script every night. The attacker could modify the shell script, hiding a small amount of code in the legitimate script in order to start their malware.

It pays to check the code and configuration files for even legitimate scheduled tasks to make sure they haven't been hijacked by an attacker. Look for any such files that have been recently modified. Perhaps you have a copy of the original files that you could compare with the jobs found on the compromised system? Linux has a tool called "diff" to compare two files and show any differences.

NETWORKING

Processes talking to other hosts?
Processes listening on unexpected ports?
Any network behavior from "suspicious" processes

```
live_response/network/netstat_-lpeanut.txt
```

Unexpected network behavior can also clue us in to suspicious processes.

Processes unexpectedly talking out to hosts on other networks are fairly easy to spot. That's why most malware chooses to "beacon"—make short connections on regular intervals—rather than maintaining persistent connections. Unless you're very lucky, tools like "netstat" are unlikely to catch the malicious process talking to its command and control server.

Just to give you an idea of what a persistent connection might look like, here is my SSH session into my lab virtual machine:

```
tcp ... 192.168.10.135:22    192.168.10.1:61166    ESTABLISHED ... 2209/sshd: ...
```

The local address of the virtual machine is 192.168.10.135 and the connection is coming from the remote host 192.168.10.1. The PID of the SSH daemon handling this connection is 2209.

It's also important to pay attention to processes listening for inbound network connections. These could be potential back doors into your system:

```
tcp ... 0.0.0.0:1337    0.0.0.0:*    LISTEN    89    638037    58719/lsof
```

lsof is not a command that should be listening on a network port, and 1337/tcp is not a standard port used by any network service that I know of.

LAB – HONEYPOT PART 2

Evil is as evil does...

How much more can we figure out about suspicious processes in our compromised honeypot? Again, my thanks to Tyler Hudak (@SecShoggoth) for providing the UAC data.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



ANALYSIS: USER CONTEXT

The user context of a process— what user the process is running as, which user owns the executable the process is running from— can provide valuable clues in an investigation. Also attackers may subvert legitimate user accounts and groups to create back doors for themselves.

IDENTITY BASICS

Users have a default User ID (UID) and Group ID (GID)

*Assigned in **/etc/passwd***

Users may belong to other groups

*As listed in **/etc/group***

Files have a user owner and a group owner

Default is the user who created the file

Internally, the Linux operating system stores user contexts in terms of numeric user ID (UID) and group ID (GID) values. The UID values are associated with specific user names in the `/etc/passwd` file:

```
root:x:0:0:root:/root:/bin/bash
sshd:x:108:65534:./run/sshd:/usr/sbin/nologin
lab:x:1000:1000:Lab User:/home/lab:/bin/bash
```

The third field is the user's UID and the fourth field is their default GID. In this example, the "sshd" user is UID 108 and has a default GID of 65534.

You can discover the group name associated with GID 65534 by looking at `/etc/group`:

```
...
plugdev:x:46:lab
staff:x:50:
games:x:60:
users:x:100:
nogroup:x:65534:
...
```

GID 65534 corresponds to the group named "nogroup".

Users have a single default GID in their `/etc/passwd` entry but can be the member of multiple other groups via the `/etc/group` file. In the example above, the "lab" user is a member of group "plugdev".

Files in the file system also have a user and group associated with them. By default, newly created files inherit the UID and default GID of the user that creates the file. The lab user shown in the `/etc/passwd` entries above will create new files owned by UID 1000 and GID 1000 by default.

The administrative user is allowed to change ownerships of files using the "chown" command. Regular users are not permitted to change ownerships.

ANALYTIC NOTES

What user is that malicious process running as?

Who owns the malicious executable?

These are often clues to the initial compromise

What other processes are running as this user?

What other files/directories do they own?

Expand the scope of your investigation

The user context of a process and/or its executable may provide valuable clues about how a given system was compromised. For example, if you see executables appearing in your file system owned by the web server user or processes running as the web server user, a reasonable theory is that a web application compromise is responsible.

Looking at the UAC data, our suspicious lsof and xterm processes are running as the "mail" user:

```
$ grep -E '(lsof|xterm)' live_response/process/ps_-ef.txt
mail      1811      1  0 16:59 pts/0      00:00:00 lsof -l -k -p 1337
mail      1817      1  0 16:59 pts/0      00:00:00 xterm
```

Are we to imagine a compromise of the email server was to blame? In reality, there is no email security issue that I am aware of. I just chose "mail" as the user to launch our "suspicious" processes to make the collected data more interesting.

You can also pivot out from this information. What other processes are running as the suspicious user? Do you find other files owned by this user?

```
$ grep mail live_response/process/ps_ef.txt
mail      1811      1  0 16:59 pts/0      00:00:00 lsof -l -k -p 1337
mail      1817      1  0 16:59 pts/0      00:00:00 xterm
mail      1818      1  0 16:59 pts/0      00:00:00 tail -f /var/log/wtmp
```

Here we can also see the "tail" process that we found earlier in the lsof output along with our suspicious "lsof" and "xterm" processes.

In some cases, you can use process start times to pick out the suspicious processes. Notice "lsof", "tail", and "xterm" were all started around the same time. You'll also see that this time is much later than the start time for the SSH daemon and other processes that are typically started at boot.

UID NOTES

Any account with UID 0 has admin rights
Normally only "root" account has UID 0

Accounts with UID < 1000 are service accounts
Should be locked
No interactive logins!

All accounts with UID 0 have administrative privileges. Under normal circumstances only the "root" account has UID 0. However, multiple accounts with the same UID are permitted in `/etc/passwd`. Sometimes you will see attackers creating extra UID 0 accounts and trying to hide them in large password files.

There is a quick bit of command line kung fu which helps here:

```
$ sort -t: -k3,3 -n /etc/passwd
root:x:0:0:root:/root:/bin/bash
toor:x:0:0:surprise!:/tmp:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
[... snip ...]
```

We are sorting the `/etc/passwd` file numerically ("-n") on the 3rd colon-delimited field ("-t: -k 3,3"), which is the UID. The extra UID 0 "toor" account that was hidden in the file is now easy to see. And since the entire file is now sorted, you'll be more likely to notice any other duplicate UID issues in the file.

Normal user accounts are typically assigned UIDs starting with UID 1000. UIDs below 1000 are reserved for service accounts like the "www-data" and "mail" users. These service accounts are typically "locked" to prevent interactive logins under these accounts:

```
root:x:0:0:root:/root:/bin/bash
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
lab:x:1000:1000:Lab User:/home/lab:/bin/bash
```

Notice that the "mail" user has "/usr/sbin/nologin" as their login shell while the other users above have "/bin/bash". The "nologin" program will log any attempt to log in as the "mail" user and then terminate the user session.

These service accounts are also locked in the /etc/shadow file, where user password hashes are stored:

```
root:$6$B9aExTQYCb2JDhm4$CN[... rest of hash not shown
...]:0:99999:7:::
mail:!:19392:0:99999:7:::
lab:$6$vTUE4GcVG6Lk/6hT$od[... rest of hash not shown
...]:0:99999:7:::
```

Accounts that allow interactive logins have long user password hashes in the second field. Locked accounts like the mail user have "*" or "LOCKED" or some other string that is not a valid password hash.

However, sometimes attackers will unlock service accounts and use them as back doors. Keep an eye out for service accounts that have "bash" or some other legit login shell instead of "noshell". Make sure their /etc/shadow entries don't have valid password hashes in field #2.

SUDO

Grants admin access on a limited basis

Configured via **/etc/sudoers**

Some groups have special Sudo privileges

Look out for unauthorized group memberships!

If you know the root password for a machine, you can get administrative access using the "su" command. These days, however, most sites prefer to use the Sudo program to grant administrative privileges on a more granular basis.

Administrative access is configured via the /etc/sudoers file. Often this file grants access based on which group(s) a given user belongs to:

```
%sudo  ALL=(ALL) ALL
```

This entry says that any user who is a member of group "sudo" can use Sudo to execute any command they want, as any user. In other words, members of this group have unlimited administrative access. Make sure to audit your /etc/group file and make sure that unauthorized users are not being added to this group!

Of course, attackers could edit the sudoers file directly and simply add an entry for their compromised account:

```
mail    ALL=(ALL) ALL
```

But adding a user to the /etc/group entry for "sudo" is typically more stealthy.

SET-ID BITS

Some processes need to run with privilege

Set-UID: run as executable owner, not user

Set-GID: group privs of executable, not user

Adding set-UID is a subtle post-exploitation back door

```
live_response/system/suid.txt  
live_response/system/sgid.txt
```

Certain programs in Linux need to run with administrative privilege. For example, if a user wants to change their password, the `/etc/shadow` file must be updated. Only the root user has the permissions to modify this file. The "passwd" program is installed "set-UID" to the root user:

```
$ ls -l /usr/bin/passwd  
-rwsr-xr-x. 1 root root 27856 Aug  9 2019 /usr/bin/passwd
```

The "s" in the first part of the permissions vector indicates that the program is "set-UID".

"Set-UID" means that when the passwd program runs, it executes with the privileges of the executable's owner— "root" in this case— rather than running as the user who executed the program. Obviously, developers need to be very careful when writing programs that will execute set-UID!

Set-GID works the same except that it changes the default group that a given process runs as. Set-GID programs are most common for queueing systems like email and printing. Users run a set-GID program that gives them rights to put new jobs into the queue, which is writable for some application-specific group. Then another process comes along and processes their job out of the queue.

If attackers breach a system and achieve admin access, they can add the set-UID bit to any program they want. A classic back-door is to add the set-UID bit to the "bash" program or one of the other Linux command shells. They may do this on the /usr/bin/bash program itself or make a copy of the program in some other directory and make that copy set-UID.

```
# cp /bin/bash /tmp/.ICEd-unix/evilsh
# chmod u+s /tmp/.ICEd-unix/evilsh
# ls -l /tmp/.ICEd-unix/evilsh
-rwsr-xr-x. 1 root root 964600 Jul  3 17:41 /tmp/.ICEd-
unix/evilsh
```

To get root access, simply invoke the set-UID shell with the "-p" option.

Look for set-ID bits appearing on executables that shouldn't have them. And look for set-ID executables appearing outside of the /usr file system. UAC creates lists of set-UID and set-GID programs found on the system. Look at live_response/system/suid.txt and .../sgid.txt in the UAC output.

AUTHORIZED_KEYS

`$HOME/.ssh/authorized_keys`

Contains public keys for SSH authentication

Popular post-compromise back door

SSH supports public-key based authentication. The user creates a public/private key pair and then places the public key into the `.ssh/authorized_keys` file in their home directory on the remote system(s) they wish to log into. When they go to log in, the remote server uses the public key from `authorized_keys` to encrypt a challenge for the user. The only way to decrypt this challenge is with the private key, which the user keeps in some secure storage on their primary machine. If the user can decrypt the challenge, then they have access to the private key, and the remote system trusts them to log in.

There are numerous post-exploitation scripts in the wild that attempt to add an attacker's public key into `authorized_keys` files for legitimate users on the system. If the attacker achieves admin access then putting the key into `/root/.ssh/authorized_keys` is ideal, because then the attacker would have root access whenever they want. But any user will do.

Note that the compromised user account must have a legit shell like `"/bin/bash"`– `"noshell"` will still block the account. However, the `/etc/shadow` entry DOES NOT need to have a valid password hash. Public key authentication supersedes password authentication on most systems.

Be sure audit users' `authorized_keys` files after a successful intrusion. UAC collects these files automatically as part of its data gathering. You'll find them under "`[root]/home/*/.`ssh" and "`[root]/root/.`ssh"

Note that if you find attackers have added entries to `authorized_keys`, it's likely that an automated post-exploitation script added the same key to all files. That means you can use the attackers' key as an IoC to locate other systems and users that have been compromised.

POST-EXPLOITATION CHECKLIST

Added **authorized_keys** entries

Additional unexpected group members

Unlocked service accounts

Extra UID 0 accounts

Direct **/etc/sudoers** modifications

Unauthorized set-UID bits

One of the easiest ways to blend in is for attackers to leverage existing accounts on the system rather than creating their own. Follow this post-exploitation checklist to look for possible back-doors.

- Look in \$HOME/.ssh/authorized_keys for unexpected entries. If you find them, run a search through your environment for other accounts with these keys.
- Audit /etc/group for users being unexpectedly added to groups— particularly groups which have special privileges in /etc/sudoers, et al.
- Make sure all service accounts have "noshell" (or similar) as their shell and DO NOT have valid password hashes in /etc/shadow.
- Look for extra UID 0 accounts in /etc/passwd. You can use this command to output only the UID 0 accounts: "**awk -F: '\$3 == 0' /etc/passwd**"
- Make sure nothing has been added to /etc/sudoers which would grant extra privileges.
- Make sure no extra set-UID/set-GID programs have appeared on the system and that the set-UID/set-GID bits have not been added to existing programs.

LAB – HONEYPOT PART 3

Bad user! No biscuit!

What does user information tell us about our honeypot compromise? As always, thanks to Tyler Hudak (@SecShoggoth) for providing the UAC data.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



BULK EXTRACTOR

We will cover detailed memory analysis in upcoming sections. However, bulk extractor can be a quick way to triage memory samples and/or to do analysis when tools like Volatility™ are not available.



THE BAD OLD DAYS

Memory analysis before Volatility was hard

Mostly looking at **strings** output

Difficult to find needles in that haystack

In the days before Volatility™ memory analysis largely consisted of running "strings" against your memory dumps and looking for something "interesting".

The problem was there tended to be a lot of garbage and irrelevant data in the output. We would write patterns to try and find important information like URLs and IP addresses. But the patterns were hard to get right and often returned junk data. And how would you decide which URLs or IPs were most relevant?

It was a frustrating "needle in a haystack" kind of problem.



BULK_EXTRACTOR

Recognizes and categorizes critical data types
URLs, hostnames and IPs, email addresses, etc

Creates histograms for each data type
Quickly see the most relevant entries

Extracts a PCAP with recent network activity
Break out your favorite network forensic tools!

Bulk extractor is a high-speed data classification and sorting tool. It has excellent pattern-matching rules to find URLs, email addresses, domains, IP addresses, and even credit card info a telephone numbers. Each type of data is extracted into its own separate file (with byte offsets so you know where in the image each piece of data was found), and then bulk extractor creates histograms to show which items appeared most frequently. Frequently hit URLs might be C2 beacons, frequent IP addresses might be the source of malicious traffic, and so on.

Bulk extractor will work on any kind of raw data. You can process disk images with it, and this is sometimes useful. When working with memory images, however, bulk extractor can recognize packets found in the memory image, and automatically extract them into a PCAP file you can process with your favorite network forensic tools. This can provide a valuable glimpse into recent network traffic to and from the system.

PROCESSING YOUR DATA

```
bulk_extractor -o outputdir memory.img
```

Default settings are good

Just specify output directory and memory image

Output directory should not already exist

Also run **strings** because context can be helpful

The bulk extractor default settings are fine. Just run the tool specifying an output directory and a data file to run against. The output directory should not already exist. If you try to write to an existing directory, bulk extractor quits with an error message.

Note that when I am processing a memory image with bulk extractor, I will still also use the "strings" command to extract ASCII strings from the image. Bulk extractor is great for finding interesting data like URLs and email addresses, but I often want to look at the context around those findings. Having the complete "strings" data lets me see what other information is found nearby to the item of interest.

Note that there is a GUI tool companion to bulk extractor called BEViewer. BEViewer is a Java GUI that you can use to run bulk extractor and examine the output. You can click on an item like a URL in BEViewer and a hex dump pane shows you that item in context in your data source. This is another good way to see the context around your evidence items.

Here is a command line session showing me processing a memory image with bulk_extractor and strings in our lab virtual machine:

```

[root@LAB bulk_extractor]$ unzip /images/All-Images/honeynet-challenge-07/victoria-v8.memdump.img.zip
Archive:  /images/All-Images/honeynet-challenge-07/victoria-...
  inflating: victoria-v8.memdump.img
[root@LAB bulk_extractor]$ bulk_extractor -o hc07 victoria-v8.memdump.img
bulk_extractor version: 1.5.5
Hostname: LAB
Input file: victoria-v8.memdump.img
Output directory: hc07
Disk Size: 268369920
Threads: 4

        [... snip ...]
Elapsed time: 12.5612 sec.
Total MB processed: 268
Overall performance: 21.3649 MBytes/sec (5.34124 MBytes/sec/thread)
Total email features found: 10513
[root@LAB bulk_extractor]$ strings -a -t d victoria-v8.memdump.img |
                           gzip >hc07/strings.asc.gz

[root@LAB bulk_extractor]$ cd hc07/
[root@LAB hc07]$ ls
aes_keys.txt                pii.txt
alerts.txt                  pii_teamviewer.txt
ccn.txt                     rar.txt
ccn_histogram.txt           report.xml
ccn_track2.txt              rfc822.txt
ccn_track2_histogram.txt    sqlite_carved.txt
domain.txt                  strings.asc.gz
domain_histogram.txt        telephone.txt
elf.txt                     telephone_histogram.txt
email.txt                   unrar_carved.txt
email_domain_histogram.txt  unzip_carved.txt
email_histogram.txt         url.txt
ether.txt                   url_facebook-address.txt
ether_histogram.txt         url_facebook-id.txt
exif.txt                    url_histogram.txt
find.txt                     url_microsoft-live.txt
find_histogram.txt          url_searches.txt
gps.txt                     url_services.txt
httplogs.txt                vcard.txt
ip.txt                      windirs.txt
ip_histogram.txt            winlnk.txt
jpeg_carved.txt             winpe.txt
json.txt                    winprefetch.txt
kml.txt                     zip.txt
packets.pcap

```

START WITH THE HISTOGRAMS

```
[root@LAB hc07]$ head url_histogram.txt
# BANNER FILE NOT PROVIDED (-b option)
# BULK_EXTRACTOR-Version: 1.5.5 ($Rev: 10844 $)
# Feature-Recorder: url
# Filename: victoria-v8.memdump.img
# Histogram-File-Version: 1.1
n=19    http://yeha.sourceforge.net/
n=18    http://rt2x00.serialmonkey.com
n=16    http://www.kernel.org/doc/man-pages/.
n=15    http://www.unicode.org/onlinedat/countries.html
n=14    http://www.w3.org/1999/xhtml
```

Hmmm, that URL looks strange...

Generally the first data I am going to look at in the bulk extractor output is the histogram files:

```
url_histogram.txt
url_services.txt
ip_histogram.txt
email_histogram.txt
email_domain_histogram.txt
domain_histogram.txt
```

"url_services.txt" is a histogram based only on the hostname portion of the URL, whereas "url_histogram.txt" sorts based on the full URI path.

We are looking for anything that seems out of the ordinary. This could be URLs with hard-coded IP addresses, or domains that look like they were created with a domain generation algorithm, etc. Note that the suspicious item doesn't necessarily have to appear a large number of times in the image. Sometimes the occasional outliers at the bottom of the histogram are the significant items.

USE STRINGS FOR MORE CONTEXT

```
[root@LAB hc07]$ zgrep -F -C3 serialmonkey.com strings.asc.gz
25289984 license=GPL
25290016 description=Ralink RT2500 USB Wireless LAN driver.
25290067 version=2.1.4
25290112 author=http://rt2x00.serialmonkey.com
25290176 srcversion=ABD1C1B1A3052F0F8B56668
25290240 alias=usb:v5A57p0260d*dc*dsc*dp*ic*isc*ip*
25290304 alias=usb:v0EB0p9020d*dc*dsc*dp*ic*isc*ip*
--
[... snip ...]
```

But until you look at the URL string in context, it's difficult to know whether it's significant or not. This is where I end up referring to the complete ASCII strings data from my image.

Here I'm using "grep" to show three lines of context ("-C3") around each hit on our suspicious domain name. As we look at the strings around our string of interest, it becomes pretty clear that this URL is simply included in documentation for a Linux wireless interface driver. This no longer seems like a big deal.

PACKET ANALYSIS TRICKS

Quick summary of network traffic:

```
tshark -n -r packets.pcap -Tfields  
-e ip.src -e tcp.srcport -e ip.dst -e tcp.dstport | sort | uniq -c
```

Extract all TCP streams into files:

```
tcpflow -r packets.pcap -o outputdir -e http
```

Bulk extractor will by default look for packet residue in your memory image and output a PCAP file called "packets.pcap" with whatever data it finds. You could load this file up in Wireshark and begin investigating, but I use a couple of tricks to quickly get an idea of what data bulk extractor managed to capture.

My first step is usually to try and get a high-level idea of who is talking to whom. The command-line "tshark" program is excellent for quickly extracting a few fields from each packet and letting me create a quick histogram with some command line kung fu:

```
[lab@LAB hc07]$ tshark -n -r packets.pcap -Tfields  
-e ip.src -e tcp.srcport -e ip.dst -e tcp.dstport | sort | uniq -c  
   1 192.168.56.1      44616   192.168.56.102   22  
 271 192.168.56.1      8888    192.168.56.102  56955  
   1 192.168.56.100      255.255.255.255  
  71 192.168.56.101    59151   192.168.56.102   25  
   3 192.168.56.102    56955   192.168.56.1     8888
```

The majority of the traffic seems to be 192.168.56.102 talking to 8888/tcp on 192.168.56.1. But there's also some 25/tcp traffic into 192.168.56.102 from 192.168.56.101. In this scenario, 192.168.56.102 is the victim machine.

But what is the content of these communications? You could manually go in with Wireshark and "Follow TCP stream", but I prefer to use "tcpflow" to extract all of the TCP streams at once:

```
[lab@LAB hc07]$ tcpflow -r packets.pcap -o flows -e http
[lab@LAB hc07]$ ls -lh flows/
total 160K
-rw-rw-r--. 1 lab lab 848 Jan 1 1970 192.168.056.001.44616-
192.168.056.102.00022
-rw-rw-r--. 1 lab lab 136K Jan 1 1970 192.168.056.101.59151-
192.168.056.102.00025
-rw-rw-r--. 1 lab lab 1.4K Jan 1 1970 192.168.056.102.56955-
192.168.056.001.08888
-rw-rw-r--. 1 lab lab 952 Jan 1 1970 192.168.056.102.56955-
192.168.056.001.08888c1
-rw-rw-r--. 1 lab lab 182M Jan 1 1970 192.168.056.102.56955-
192.168.056.001.08888c2
-rw-rw-r--. 1 lab lab 5.6K Jul 5 21:16 report.xml
[lab@LAB hc07]$ ls -lh packets.pcap
-rw-r--r--. 1 lab lab 133K Jul 5 18:32 packets.pcap
```

The flows are identified by their source and destination IP address and port numbers. Since the packet data is fragmentary, we have multiple chunks of the 8888/tcp traffic rather than a single continuous stream.

Note that if there were any HTTP sessions in the PCAP file, the "-e http" switch to tcpflow would have created a *-HTTP file for each HTTP flow that would extract the downloaded object(s) from the HTTP application-layer protocol. Unfortunately, with PCAP data extracted from RAM, it's rare for you to capture a complete enough HTTP session for this to work.

Look at the relative file sizes in the above output. "packets.pcap" is only 133K, but the 25/tcp flow is 136K and the last chunk of 8888/tcp traffic is 182MB! Clearly "tcpflow" is having some trouble with the fragmentary data bulk extractor pulled from memory.

The 25/tcp traffic is nothing but long strings of 'A' characters. This makes me wonder if it is somebody trying some sort of buffer overflow against the email server on 192.168.56.102.

The first chunk of 8888/tcp traffic looks like documentation from the "distutils" package and the second chunk is apparently a list of checksums. Possibly a package download? The final chunk is all nulls, but clearly erroneous since it is 1000x the size of our original PCAP file.

The 22/tcp traffic is short, but you can see an exchange of encryption algorithms from the early stages of an SSH connection.

LAB – BULK EXTRACTOR

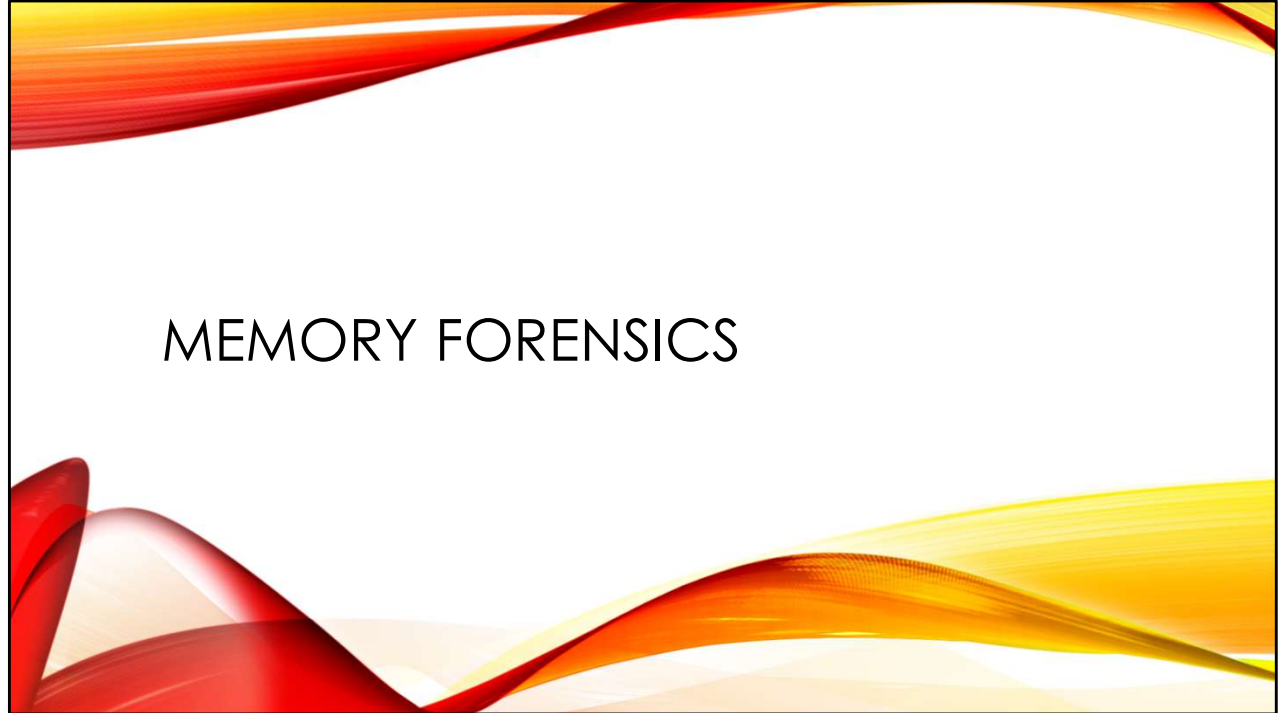
Quick and dirty memory triage

We have memory from an actual compromised system, but lack the other dependencies to use Volatility for analysis. We can still generate some interesting leads, however. Thanks to Tyler Hudak (@SecShoggoth) for providing the memory sample we will be using for this lab!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Memory analysis is a powerful forensic technique. But there are some unique complications when it comes to doing memory forensics on Linux.



WHY MEMORY FORENSICS?

Size matters – faster acquisition/analysis, less storage

See more –

- Cached file information

- Volatile process, network data

- Rootkit indicators

- Encryption keys

Memory analysis is a key forensic technique for all types of investigations. Disks continue to get larger and larger, making traditional "dead box" analysis less and less practical. It is much easier to collect, analyze, and store 64GB of RAM compared to 2 Terabytes of disk.

Key artifacts for your investigation can be found in memory. Process executables, shared libraries, program data are all stored in a structured fashion in memory. Network connections are tracked. Even encryption keys are available. Aggressive caching and "memory mapping" of files allows the investigator to find many file system artifacts. Rootkit hiding techniques, process injection and hollowing, and other types of malicious activity become obvious with the right memory analysis tools.

There's no question that memory analysis has greatly improved the lives of forensic investigators. But Linux presents some unique challenges for memory analysis.



TOO MANY KERNELS!

Volatility needs kernel version/distro specific "profile"

Dependencies for creating profile may not be available

No matter what the operating system, Volatility needs a kernel-specific "profile" in order to be able to analyze a memory dump. This profile provides memory offsets and other information for where Volatility should look for critical information in the memory dump.

On Windows and Mac, this isn't as much of an issue because each version of those operating systems uses a single kernel. Everybody running "Windows 11 22H2" or "MacOS 13.3" is using the same kernel. Once somebody creates a Volatility profile for the latest Windows or MacOS version, then anybody else can use that profile to analyze memory dumps from those systems.

The Linux world is MUCH more fragmented. Even within a single vendor's Linux release, say "RHEL 9.1", there can be dozens of different kernel versions in use, depending on when a machine last got its kernel updates. Adding to this, each vendor compiles their own kernel with different options, so even if we had kernels with the same version numbers from RHEL and Ubuntu, we would need different Volatility profiles to analyze them. For a single investigation you might need dozens of different Volatility profiles to cover all of the different Linux kernels you have captured memory from.

For Volatility v3, creating a Linux profile requires running the `dwarf2json` program against a copy of the Linux kernel that was compiled with debugging symbols and not stripped after compilation. These "debugging kernels" are not usually installed with new system images and getting them generally means finding the `-debug` or `-dbg` version of a given kernel package, e.g. `linux-image-5.10.0-21-amd64-dbg`. The packages are also generally quite large— multiple gigabytes in size when unpacked.

And some Linux distros don't make debugging kernels available at all (Red Hat operating systems seem to be particularly guilty of this). Or you may be dealing with a memory image from a Linux version that has reached end of life and packages are no longer available for the operating system. Even the Volatility documentation admits "it may not be possible to find the right symbols to analyze a Linux memory image with Volatility" (<https://volatility3.readthedocs.io/en/latest/symbol-tables.html>).

There is a crowdsourcing project to build a library of Volatility profiles for common Linux versions. You can check <https://isf-server.techanarchy.net/> to see if a profile is available for the Linux version you want to analyze.

Things were not better in the Volatility v2 days. Volatility memory analysis still required creating a kernel version specific profile. For Volatility v2, this was accomplished by compiling the program in the kernel build environment for the given OS version. Because the system being analyzed might be the only system with that particular kernel version, we often ended up in this weird place of having to compile the program necessary for profile creation *on the system we were trying to analyze*. And if that system lacked the necessary build environment, we were generally out of luck.

FIND KERNEL, BUILD PROFILE

```
$ find / -name vmlinu\* -size +100M 2>/dev/null
/usr/lib/debug/boot/vmlinux-5.10.0-21-amd64
$ file /usr/lib/debug/boot/vmlinux-5.10.0-21-amd64
/usr/lib/debug/boot/vmlinux-5.10.0-21-amd64: ELF 64-bit LSB executable, x86-
64, version 1 (SYSV), statically linked, BuildID[sha1]=5e5d3209033f927baa64...,
with debug info, not stripped
$ ls -lh /usr/lib/debug/boot/vmlinux-5.10.0-21-amd64
-rw-r--r-- 1 root root 627M Jan 21 14:35 /usr/lib/debug/boot/vmlinux-5.10.0-...
$ dwarf2json linux --elf /usr/lib/debug/boot/vmlinux-5.10.0-21-amd64
>vmlinux-5.10.0-21-amd64.json
$ mkdir -p /usr/local/volatility/volatility3/symbols/linux
$ cp vmlinux-5.10.0-21-amd64.json
  /usr/local/volatility/volatility3/symbols/linux
```

If you're very lucky, a debugging kernel may already be present in the image that you're analyzing. Different Linux distributions will put this kernel in different directory locations—you see `/usr/lib/debug` in this example, but I've also found them under `/lib/modules` and other places. My trick here is to locate large files with a name like "vmlinux" (uncompressed image) or "vmlinuz" (compressed image). Normal kernels that have been stripped are typically smaller than 10MB, so finding a kernel larger than 100MB generally guarantees you've found a debugging kernel.

We can verify this with the "file" command. In the output look for phrases like "with debug info" or "debugging info" and "not stripped". Also note the size of the kernel image we've found— 627MB!

Once we locate an appropriate kernel, we need to run `dwarf2json` against it. `dwarf2json` is not a standard Linux application. Download the source from <https://github.com/volatilityfoundation/dwarf2json> and compile it. `dwarf2json` is written in GoLang, so you will likely need to download other dependencies to build the binary. FYI `dwarf2json` requires a minimum of 16GB of RAM to run.

Finished profiles are kept in the Volatility installation directory under `.../volatility3/symbols/linux` but later I'll show you another installation option that lets you keep the profile in the same directory with the memory image.

MEMORY ACQUISITION TOOLS

AVML – Free, file output only

LiME – Free, kernel driver, output to file or network

F-Response – Costs money, agent for disk/memory access

Currently the easiest way to acquire Linux memory is with the free AVML ("Acquire Volatile Memory Linux") tool from Microsoft. You will likely need to build it from source code (<https://github.com/microsoft/avml>) but once you have the binary you can run it on any Linux system to extract a memory dump to the local file system. Because Microsoft wrote the tool to work in their Azure cloud, there are additional options to copy the memory dump to an Azure blob store or just PUT the memory dump to a URL you specify.

Memory acquisition technology in Linux has been a strange journey. In the 1990s you could acquire memory in Linux just by dumping the `/dev/mem` device with a tool like `dd`. But the Linux kernel developers decided (probably correctly) that this direct memory access was of more help to the attackers than the defenders. So they deliberately limited the amount of memory you could access through `/dev/mem`.

That meant bringing your own kernel driver if you wanted to access all of memory on a Linux system. Joe Sylve's LiME (Linux Memory Extractor) is still a popular driver-based solution for accessing RAM. It works on both standard Linux servers and Android, and has the ability to write memory images over the network and not just to local files. You can download the source from <https://github.com/504ensicsLabs/LiME>

However, the problem with driver-based solutions is that you have to build the driver using the kernel build environment for the specific kernel version where the driver must be installed. And as was the case with building Volatility profiles, this might not be possible due to missing dependencies.

Instead of loading a kernel driver, AVML interrogates the `/proc/kcore` device found in most Linux distributions (`/proc/kcore` is a memory access path for debugging live systems). This `/proc/kcore` technique was originally developed by an earlier Linux memory acquisition tool called LinPmem, which is no longer supported.

F-Response (<https://www.f-response.com/>) is a commercial, agent-based solution that can be deployed as needed to extract memory remotely over the network. It brings its own kernel driver along to access memory.

FIFOS FTW!

```
# mkfifo /tmp/myfifo
# cat /tmp/myfifo | nc -w 1 remotehost 9999 &
[1] 21039
# avml /tmp/myfifo
```

AVML is easy to use but it insists on writing the memory dump to a local file. This problem is that this may overwrite evidence in unallocated blocks in the file system.

A FIFO is like a pipe on the command line that connects the output of one program to the input of another. But a FIFO does this via an object that looks like a file in the file system, so we can use it to fake out AVML:

1. First we create the FIFO with the `mkfifo` command. The FIFO uses a single inode in the file system but consumes no disk blocks.
2. Next we start a command reading from the FIFO. I'm using netcat in the example to push the data over the network to some remote system. You will previously have had to set up a listener on that system to receive the memory dump (on the remote system: `"nc -l -p 9999 >memory-image.lime"`).
3. Finally, run AVML and tell it to write the output to the FIFO.

Netcat will automatically close down the network connection one second after the memory dump is complete (that's the `"-w 1"` option in the example above). You are welcome to use any other tool to move the memory dump over the network, but netcat is typically installed by default in most Linux distributions.

LAB – MEMORY ACQUISITION

And remember this is your best case scenario!

Time to get some practical experience collecting Linux memory and building Volatility profiles. I have installed all of the necessary dependencies in the lab virtual machine. You're welcome.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Now we're going to look at some of the more useful Volatility plugins for Linux analysis.

VOLATILITY BASICS

Diagram illustrating the Volatility command structure and output:

Annotations:

- Profile location: `-s .`
- Memory image file: `-f memory.lime`
- Volatility plugin choice: `linux.pslist.PsList`

```
$ vol3.py -s . -f memory.lime linux.pslist.PsList
```

OFFSET (V)	PID	TID	PPID	COMM
0x93f4c01f17c0	1	1	0	systemd
0x93f4c01f5f00	2	2	0	kthreadd
...				

Earlier I noted that Volatility looks for Linux profile files in the `.../volatility3/symbols/linux` directory under whatever directory you installed Volatility into. You can specify an alternate directory to look for profiles with the `"-s"` flag as we're doing here—`"-s ."` means look in the current directory. This allows you to keep your Volatility profile together with your memory image in the same directory, which may be more convenient. Use `"-f"` to specify the memory image file to work on.

Finally, choose the Volatility plugin you want to run. You can get a list of all available Linux plugins from the `--help` option:

```
$ vol3.py --help | grep linux.
```

<code>banners.Banners</code>	Attempts to identify potential linux banners in an
<code>linux.bash.Bash</code>	Recovers bash command history from memory.
...	

The `PsList` plugin is a good one to use for testing. It runs quickly and if you get useful output you will know that you have a good memory dump and profile.

WHICH KERNEL VERSION?

```
$ vol3.py -f memory.lime banners.Banners
Progress: 100.00          PDB scanning finished
Offset  Banner

0x150d85fa2      Linux version 5.10.0-21-amd64 (debian-kernel@lists.debian.org)
(gcc-10 (Debian 10.2.1-6) 10.2.
0x264a00200      Linux version 5.10.0-21-amd64 (debian-kernel@lists.debian.org)
(gcc-10 (Debian 10.2.1-6) 10.2.1 20210110, GNU ld (GNU Binutils for Debian)
2.35.2) #1 SMP Debian 5.10.162-1 (2023-01-21)
0x28cb7e020      Linux version 5.10.0-21-amd64 (debian-kernel@lists.debian.org)
(gcc-10 (Debian 10.2.1-6) 10.2.
...
```

If you're handed a Linux memory image without a profile, you will need to figure out the version of the Linux kernel that was running when the memory image was taken. The `Banners` plugin will search through the memory image for Linux kernel banners. Hopefully this will give you enough clues to begin tracking down the debugging kernel you need to build a profile to analyze the memory image.

Sometimes looking for strings in the memory image can provide additional useful information:

```
$ strings -a memory.lime | grep PRETTY_NAME=
PRETTY_NAME=
PRETTY_NAME=%s
PRETTY_NAME=
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
...
```

Here we can see the popular name for the Linux distribution the memory image was taken from. This line comes from the operating system's `/etc/os-release` file.

PROCESS INFO

linux.psaux.PsAux – full command lines

linux.pslist.PsList – image offsets

linux.pstree.PsTree – process hierarchies

There are multiple volatility plugins for looking at process information:

- `linux.psaux.PsAux` gives process ID and parent process ID along with the full command-line.
- `linux.pslist.PsList` just gives the basic command name, PID, and PPID but also includes the virtual memory offset of the process structure. This may be useful if you want to do a deeper dive into a specific process.
- `linux.pstree.PsTree` shows a hierarchical view of processes. While this isn't as useful on Linux as it is on Windows, it can help you determine how a given process was started.

DIGGING DEEPER

linux.lsof.Lsof – list files a process has open

linux.elfs.Elfs – process binary and shared libs

linux.proc.Maps – all process memory segments

When we were working with UAL earlier, we saw the output from the `lsOf` command. Volatility can extract similar information from the memory dump:

```
$ vol3.py -q -s . -f memory-avml.lime linux.pslist.PsList | grep avml
0x93de88e75f00 2190 2190 1325 avml
$ vol3.py -q -s . -f memory-avml.lime linux.lsof.Lsof --pid 2190
PID      Process FD      Path
2190     avml    0      /dev/pts/0
2190     avml    1      /dev/pts/0
2190     avml    2      /dev/pts/0
2190     avml    3      /proc/kcore
2190     avml    4      /images/memory/memory/memory-avml.lime
```

If you don't provide the `--pid` option, the `Lsof` plugin will dump this information for all processes on the system. This could be useful if you're looking for suspicious paths and hidden directories.

The `Elfs` plugin will show you the executable the process is running from and any shared libraries it is using:

```
$ vol3.py -q -s . -f memory-avml.lime linux.elfs.Elfs --pid 1325
```

PID	Process	Start	End	File Path
1325	bash	0x5569e180e000	0x5569e183c000	/usr/usr/bin/usr/bin/bash
1325	bash	0x7ff921c45000	0x7ff921c67000	/usr/usr/lib/usr/lib/x86_64-linux-gnu/usr/lib/x86_64-linux-gnu/libc-2.31.so
1325	bash	0x7ff921e1a000	0x7ff921e1b000	/usr/usr/lib/usr/lib/x86_64-linux-gnu/usr/lib/x86_64-linux-gnu/libdl-2.31.so
1325	bash	0x7ff921e20000	0x7ff921e2e000	/usr/usr/lib/usr/lib/x86_64-linux-gnu/usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
1325	bash	0x7ff921e58000	0x7ff921e5b000	/usr/usr/lib/usr/lib/x86_64-linux-gnu/usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
1325	bash	0x7ff921e73000	0x7ff921e74000	/usr/usr/lib/usr/lib/x86_64-linux-gnu/usr/lib/x86_64-linux-gnu/ld-2.31.so
1325	bash	0x7ffe8c545000	0x7ffe8c547000	[vdso]

The path names in the output are clearly messed up. Hopefully this output bug will be fixed in future releases. Still the output is useful for finding executables and libraries in strange directories.

The `Maps` plugin provides even more detailed information about all of the memory segments the process is using. This is where you might see reflexively loaded memory segments and other evil.

BASIC NETWORK INFO

```

root@LAB:/images/memory# vol3.py -q -s -f memory-avml.lime linux.sockstat.Sockstat | grep AF_INET
4026531992 538 12 0x93de62af8440 AF_INET DGRAM UDP 0.0.0.0 5353 0.0.0.0 0 UNCONNECTED -
4026531992 538 13 0x93de456cf300 AF_INET6 DGRAM UDP :: 5353 :: 0 UNCONNECTED -
4026531992 538 14 0x93de62affb40 AF_INET DGRAM UDP 0.0.0.0 59162 0.0.0.0 0 UNCONNECTED -
4026531992 538 15 0x93de456cc600 AF_INET6 DGRAM UDP :: 57885 :: 0 UNCONNECTED -
4026531992 542 21 0x93de45cf0000 AF_INET6 RAW ICMPv6 :: 58 :: 0 UNCONNECTED -
4026531992 542 23 0x93de4436e600 AF_INET DGRAM UDP 192.168.10.137 68 192.168.10.254 67 UNCONNECTED fi1
4026531992 633 6 0x93de46c05580 AF_INET6 STREAM TCP ::1 631 :: 0 LISTEN -
4026531992 633 7 0x93de46c0abc0 AF_INET STREAM TCP 127.0.0.1 631 0.0.0.0 0 LISTEN -
4026531992 653 3 0x93de46b5bd40 AF_INET STREAM TCP 0.0.0.0 22 0.0.0.0 0 LISTEN -
4026531992 653 4 0x93de46b60000 AF_INET6 STREAM TCP :: 22 :: 0 LISTEN -
4026531992 667 7 0x93de60e63b80 AF_INET DGRAM UDP 0.0.0.0 631 0.0.0.0 0 UNCONNECTED -
4026531992 1019 4 0x93de4726bd40 AF_INET STREAM TCP 127.0.0.1 25 0.0.0.0 0 LISTEN -
4026531992 1019 5 0x93de47278000 AF_INET6 STREAM TCP ::1 25 :: 0 LISTEN -
4026531992 1197 4 0x93de46b5cec0 AF_INET STREAM TCP 192.168.10.137 22 192.168.10.1 49688 ESTABLISHED -
4026531992 1229 4 0x93de46b5cec0 AF_INET STREAM TCP 192.168.10.137 22 192.168.10.1 49688 ESTABLISHED -
4026531992 1655 20 0x93de46b08500 AF_INET6 DGRAM UDP :: 1716 :: 0 UNCONNECTED -
4026531992 1655 21 0x93de46c01300 AF_INET6 STREAM TCP :: 1716 :: 0 LISTEN -
4026531992 1692 19 0x93de5cf66900 AF_INET STREAM TCP 192.168.10.137 33058 146.75.78.49 443 ESTABLISHED -

```

The Sockstat plugin shows detail about network connections, similar to the `netstat` output we looked at with UAL. However, like `netstat` it also dumps information about Unix domain sockets which is usually much less interesting. So here I'm using "`grep AF_INET`" to pull out just the network-related information.

COMMAND HISTORY

```
# vol3.py -s . -f memory-avml.lime linux.bash.Bash --pid 1325
```

PID	Process	CommandTime	Command
...			
1325	bash	2023-02-04 21:33:24.000000	df -h
1325	bash	2023-02-04 21:33:24.000000	reboot
1325	bash	2023-02-04 21:33:54.000000	mkdir /images/memory
1325	bash	2023-02-04 21:33:56.000000	df -h /images/memory
1325	bash	2023-02-04 21:34:09.000000	avml /images/memory/...

The `Bash` plugin lets you extract command history from active shell processes in the memory image.

When a bash shell is started, it reads the saved history from the `bash_history` file in the user's home directory. You can see these commands in the first part of the output—there may be 500 lines (the default `bash_history` length) or more of output all with the same timestamp. This is the time that the shell was started and read the `bash_history`. The commands that come after, with differing timestamps, are the commands that were typed in this shell session.

We will discuss `bash_history` forensics (and anti-forensics) in more detail later in this course.

WHY IS THIS BETTER?

Command history only written to disk when shell exits
In-memory history has all commands for session

bash_history commands are not normally timestamped
*Full timestamp information visible with **linux_bash***

The Bash plugin output is more valuable than looking at the `bash_history` on disk because:

1. Command history is only saved to disk when the shell exits. So the command history in memory contains commands for the current session that have not yet been written to the `bash_history` file on disk.
2. The Bash plugin output shows the timestamp for each command. By default `bash_history` does not contain timestamps.

LAB - VOLATILITY

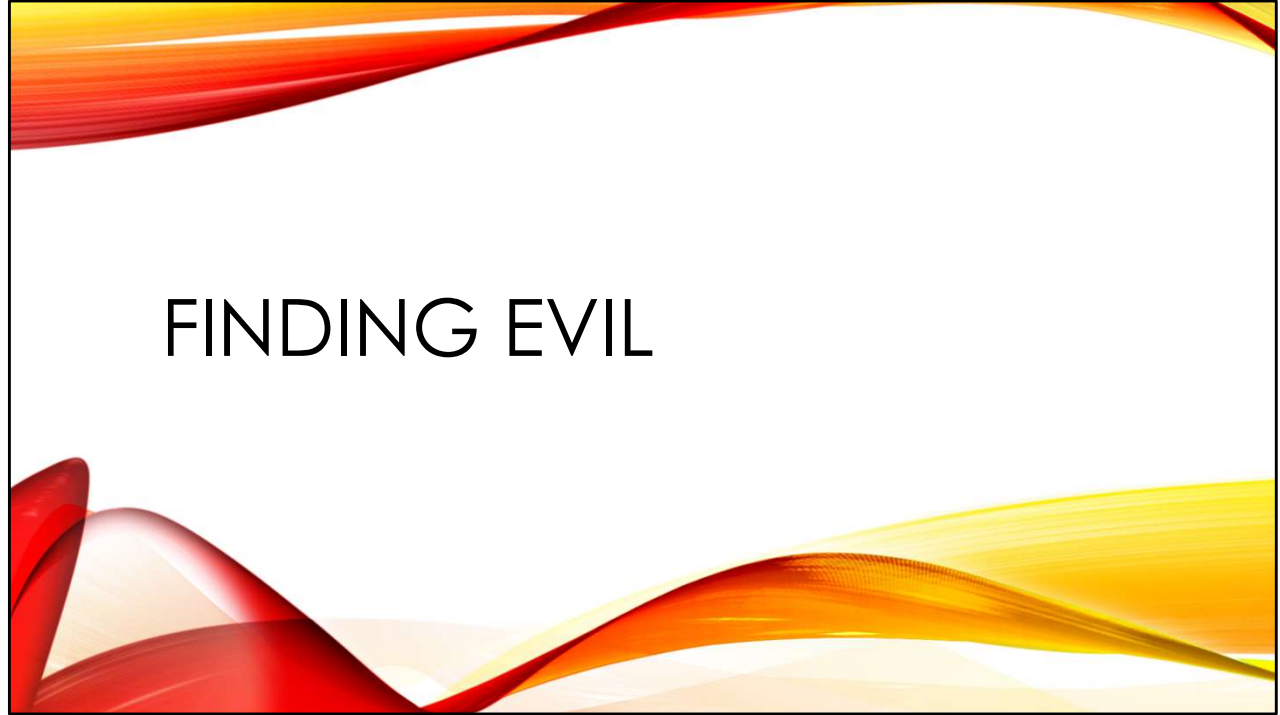
It's more fun if you do it yourself!

Experiment with Volatility plugins yourself. There are a lot of interesting artifacts to look at!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Volatility has always had a focus on finding malicious software and rootkits in memory. The Linux plugins are no different.



LINUX ROOTKITS

Linux rootkits can leverage Loadable Kernel Modules (LKM)

Volatility can help locate them!

Two types of rootkits seem to be popular in the Linux environment these days. LD_PRELOAD style rootkits install malicious shared libraries in user space to help the attacker hide files and processes. Loadable Kernel Module (LKM) rootkits use a malicious kernel module.

Volatility has some plugins that can help us spot the malicious modules used by LKM rootkits.

SPOT THAT MALICIOUS MODULE

```
$ vol3.py -s . -f memory.lime linux.check_modules.Check_modules
```

```
Module Address  Module Name
```

```
0xfffffc0e1a0c0  diamorphine
```

```
0xfffffc0b88b40  vmw_vsock_virtio_transport_commo
```

Evil Module

False positive

LKM rootkits often attempt to hide their modules so that they don't show up in the output of `lsmod` and tip off the system owner. The Linux kernel tracks loaded kernel modules several different ways, however. The `Check_modules` plugin compares the various module lists in the kernel and shows modules that are not properly tracked across all of them.

Here we see `Check_modules` flagging the Diamorphine LKM that I added to the memory image (for more information on Diamorphine see <https://github.com/m0nad/Diamorphine>). However, it also flags another module which is not malicious, but which is apparently also not registered appropriately.

Note that sometimes rootkits will not hide their LKM, but instead try to camouflage themselves using an innocuous name. The `Lsmod` plugin can be used to dump all (non-hidden) module names. You might compare the output of `Lsmod` from a "known good" system with the module list from the suspect machine to try and locate the evil module.

SPOT THE MISSING SYSCALLS

```
$ vol3.py -s . -f memory-rootkit.lime linux.check_syscall.Check_syscall |  
    awk '{print $5}' >syscalls-rootkit  
  
$ vol3.py -s . -f memory-clean.lime linux.check_syscall.Check_syscall |  
    awk '{print $5}' >syscalls-clean  
  
$ sort syscalls-* | uniq -u  
__x64_sys_getdents  
__x64_sys_getdents64  
__x64_sys_kill
```

Once the malicious kernel module is loaded, it needs to intercept legitimate system calls— a process typically referred to as *hooking*. Volatility has the `Check_idt` and `Check_syscall` modules to look for these hooks.

Syscall hooks are the usual practice for LKM rootkits. The malicious kernel module replaces the address of the normal system call with the address of its own modified system call. At the moment, the `Check_syscall` plugin simply doesn't report the hooked system calls. Here we're comparing the syscall lists from an infected memory image and an uninfected image and looking for the syscall names that only appear once between the two lists. The hooked system calls appear in the list from the clean system, but not the infected machine.

Understanding which functions are hooked can help you understand the rootkit's functionality. The `getdents` interface is used for getting directory information. Hooking this function allows the rootkit to hide files and directories. Since Linux makes process information available through the `/proc` file system, this hook can also be used to hide processes.

Diamorphine hooks `kill` because the `kill` command is used as the administrative interface for the rootkit. Sending different numeric signals to processes via the `kill` command can hide/unhide processes and elevate a process' privilege level.



KEEP ON PIVOTING!

YaraScan leverages your existing IoCs
Could also run Yara independent of Volatility

Use Bash plugin to check command history
Sometimes adversary isn't careful

Don't forget bulk_extractor and strings
Low tech but effective

Volatility has a `YaraScan` plugin that allows you to use existing Indicators of Compromise (IoCs) written as Yara signatures into your memory analysis. The only advantage the `YaraScan` plugin has over using Yara directly is that the `YaraScan` plugin outputs virtual memory offset information for where the hit was found.

The `Bash` plugin might show you command history from the attacker installing their rootkit. Even if the attacker's shell is not still active in the memory dump, the attacker's commands could have been saved to `.bash_history` and loaded into a later shell.

Also don't forget what we learned previously about using `bulk_extractor` and `strings` to pull information out of the memory dump. Some times simple tools can help.

OTHER MODULES

linux.check_afinfo.Check_afinfo

Manipulating network structs to hide

linux.tty_check.tty_check

Look for a particular keylogging method

linux.keyboard_notifiers.Keyboard_notifiers

Another keylogging method

linux.check_creds.Check_creds

Looks for process credential stealing

Volatility includes several other plugins that can be used to detect various rootkit behaviors. For more details see the Volatility command reference at:

<https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference>

In some cases, the command reference contains links to blog posts and other external documents showing how to integrate these plugins into your investigative workflow.

For some additional examples of using Volatility to discover Linux rootkits and malware, see:

<http://volatility-labs.blogspot.com/2012/09/movp-15-kbeast-rootkit-detecting-hidden.html>

<http://volatility-labs.blogspot.com/2012/09/movp-24-analyzing-jynx-rootkit-and.html>

<http://volatility-labs.blogspot.com/2012/09/movp-25-investigating-in-memory-network.html>

<http://volatility-labs.blogspot.com/2012/09/movp-35-analyzing-2008-dfrws-challenge.html>

<http://volatility-labs.blogspot.com/2012/10/phalanx-2-revealed-using-volatility-to.html>

LAB – ROOTKIT!

Always be pivoting!

How might you investigate a system if you suspect a rootkit? We'll follow a chain of artifacts to shed some light on the situation.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Because memory capture and analysis can be difficult on Linux, it's good to have a solid foundation in disk-based analysis.

Several of the disk images used in this class were created by Ali Hadi (@binaryz0ne) and his team at Champlain College for a workshop at OSDCon19. They were gracious enough to allow me to use them in this course as well. The images and their OSDCon presentation are on the course USB and also available from <https://github.com/ashemery/LinuxForensics>

DISK ACQUISITION SCENARIOS

Public Cloud

Follow vendor procedures

Private Cloud

Snapshot and copy (**qemu-img** to translate)

Local Device

ewfacquire

dc3dd

The best advice I can give about disk acquisition is to stay flexible and give yourself as many options as possible. No two cases are going to be alike and no solution is going to work for every case.

In the public cloud, each provider generally publishes guidelines for how to get a disk image of your instance. For example, here is some guidance from Amazon:
<https://aws.amazon.com/mp/scenarios/security/forensics/>

If you are running your own hypervisor, the easiest solution is to snapshot the guest you wish to forensicate. This should also get you a memory dump to analyze, in addition to the disk. However, in order to analyze the disk image from the snapshot with your forensic software you may have to convert it into a raw disk image. The `qemu-img` program is an excellent tool for converting various virtual disk formats to raw.

If you are trying to image a physical device, free capture tools include ewfacquire (writes compressed E01s) and dc3dd (raw images). Access Data also makes a free command-line tool available for acquisition on Linux systems (<https://accessdata.com/product-download>).

DIFFICULT DISK GEOMETRIES

Linux wants raw, not E01/AFF/split raw/VMDK...

Layers of configuration confuse commercial forensic tools

- Encrypted volumes

- Software RAID

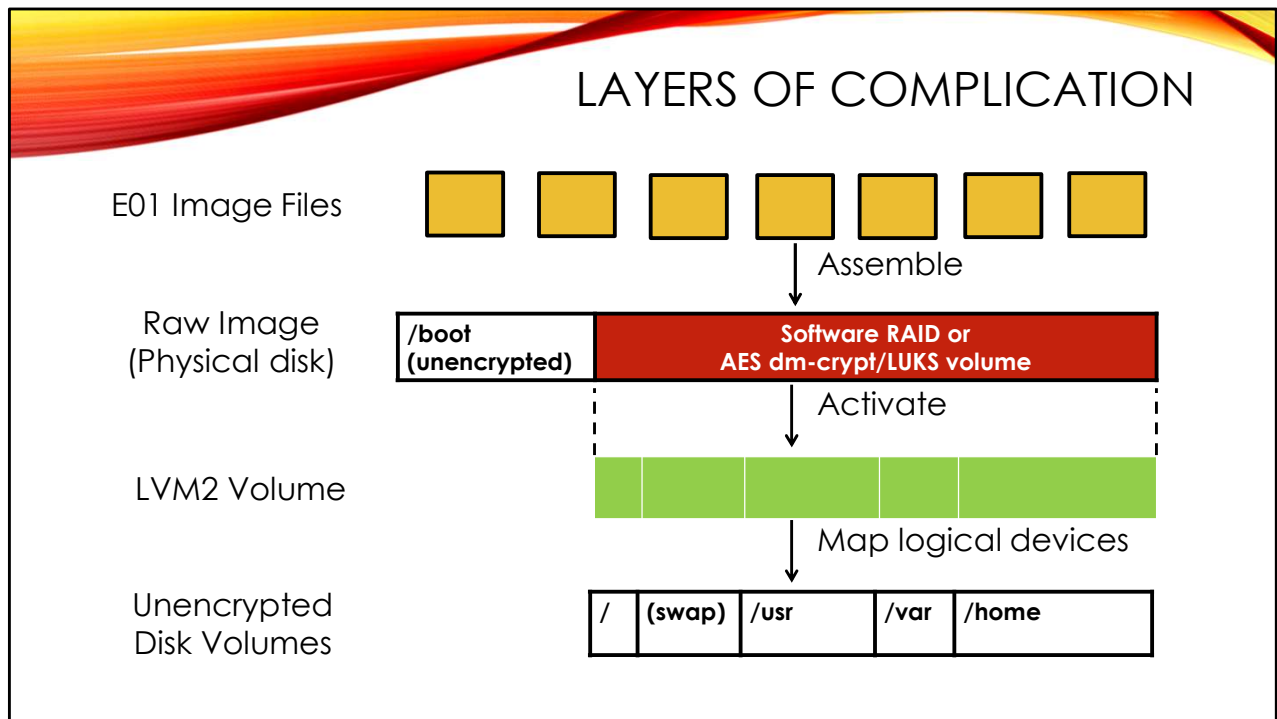
- Logical volume management

File systems often “dirty” (underplayed)

If you are analyzing disk images with the Linux and Open Source forensic toolchain, then the images generally need to be in raw form. Common forensic formats such as E01, AFF, and even split raw are not directly usable by many Linux commands. Conversion utilities like libewf (<https://github.com/libyal/libewf>) and afflib (<https://github.com/sshock/AFFLIBv3> -- supports AFF and split raw) can help, as we will see in a moment.

Once you have a raw disk image, however, the fun is only just beginning. Linux file systems are often encapsulated within additional layers of complexity, including Linux’s built-in disk encryption system (dm-crypt and LUKS) and software RAID capabilities. Linux Logical Volume Management (LVM) is a very common “soft partitioning” scheme that allows file systems to be resized on the fly.

Forensic disk imaging rarely involves gracefully shutting down the system to be acquired (shutdowns change the state of the machine). This often results in file systems that are “dirty” (underplayed)—meaning they have consistency issues that must be resolved when the file system is mounted. Mounting such file systems in read-only mode for forensics can be challenging, but there are work-arounds.



Consider a typical scenario:

1. You are given E01 files that somehow need to become a raw disk image that you can analyze. We will use libewf for this.
2. The raw disk image contains a small unencrypted `/boot` file system, but the majority of the disk is an encrypted volume or part of a multi-disk software RAID set that you need to get through (using Linux command-line tools or specialized forensic software). Or it's possible that none of this is in play—proceed to the next layer.
3. The next layer is typically multiple volumes being managed via Linux LVM (although again this is optional). Linux command-line tools can help here.
4. Each volume in the LVM configuration is typically a mountable Linux file system. Or it could be a raw Linux swap partition.

Complicated, right? Let's walk through it.

DEALING WITH E01

```
# ls
case1-webserver_meta.sqlite  Webserver.E01      Webserver.E01.txt
case1-webserver_meta.xml    Webserver.E01.csv

# mkdir -p /mnt/test/img
# ewfmount Webserver.E01 /mnt/test/img
ewfmount 20140608

# ls -lh /mnt/test/img
total 0
-r--r--r-- 1 root root 32G Feb 16 18:21 ewf1
```

The first step is to get your E01 image into something that looks like a raw file system. libewf includes a virtual file system driver (via the Linux File System in User Space or “FUSE” subsystem) that can create what appears to be a raw disk image from a collection of E01s.

First change directories to where your E01 file(s) are located. You will need to directory to mount your virtual file into— here I’m making a target directory called `/mnt/test/img`. Give the `ewfmount` command the name of the first E01 file in your collection (it automatically finds any additional segments) and the path to your target directory.

After the `ewfmount` command runs, the target directory should appear to contain a raw disk image file which is the same size as the original disk. The file name is always “`ewf1`” and it is strictly read-only.

What is actually happening here is that the `ewfmount` command is running in the background, pulling data out of the E01 files as you read from the virtual “`ewf1`” file. Yes, there is some overhead for doing things this way, and that will effect the speed at which data can be read. But it’s easier than manually converting all your E01s to raw disk images and wasting all the disk space required to hold the raw format.

WHAT'S IN THE IMAGE?

```
# mmls /mnt/test/img/ewf1
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

Probably /boot

	Slot	Start	End	Length	Description
00:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
01:	-----	0000000000	0000002047	0000002048	Unallocated
02:	00:00	0000002048	0000499711	0000497664	Linux (0x83)
03:	-----	0000499712	0000501759	0000002048	Unallocated
04:	Meta	0000501758	0066064383	0065562626	DOS Extended (0x05)
05:	Meta	0000501758	0000501758	0000000001	Extended Table (#1)
06:	01:00	0000501760	0066064383	0065562624	Linux LVM (0x8e)
07:	-----	0066064384	0066064607	0000000224	Unallocated

Now that ewfmount has given us a raw disk image, let's see what's inside!

Here I am using `mmls` from the Sleuthkit (sleuthkit.org) to dump the partition table. Although this image uses an old DOS-style partition table, `mmls` can also decode GPT and a variety of other formats automatically.

`mmls` shows a small Linux file system at the front of the disk and a larger Linux LVM partition in a DOS-style extended partition (there are no signs of full disk encryption or software RAID—hooray!). This is very typical for Linux. The small file system is `/boot`, which contains everything necessary to bootstrap the OS kernel and get things running. Once the OS is up and running, Linux automatically decodes the LVM configuration. Unfortunately, we're going to have to do that step manually.

MORE DETAIL

```
# fsstat -o 2048 /mnt/test/img/ewf1
FILE SYSTEM INFORMATION
-----
File System Type: Ext2
Volume Name:
Volume ID: 1e860db5dd43e2934d499ba1013b8832

Last Written at: 2019-10-05 05:41:51 (EDT)
Last Checked at: 2016-04-03 12:05:47 (EDT)

Last Mounted at: 2019-10-05 05:41:51 (EDT)
Unmounted Improperly
Last mounted on: /boot
...
```

We can get more detail on the small Linux file system at the front of the disk using the Sleuthkit's `fsstat` tool. Like all Sleuthkit commands, `fsstat` accepts the “-o” flag to specify a sector offset in the disk image where the file system begins. The sector offset is the “Start” column data in the `mmls` output on the previous slide.

`fsstat` tells us the type of file system we are dealing with— EXT2 in this case. We also can see where the file system was last mounted. As we suspected, this is `/boot`. The `fsstat` output also shows things like the last mounted time and whether the file system is clean or dirty. “Unmounted Improperly” means the file system is dirty.

SETTING UP A LOOPBACK DEVICE

-r is "read only"
-f is first available

Need byte offset
(sector data from *mm1s*)

```
# losetup -rf -o $((501760*512)) /mnt/test/img/ewf1
# losetup -a
/dev/loop0: [0020]:2 (/mnt/test/img/ewf1), offset 256901120
# file -s /dev/loop0
/dev/loop0: LVM2 PV (Linux Logical Volume Manager), UUID: SA3YA1-
91Rk-W5FA-cQGz-TnXl-J4yN-awbQjd, size: 33568063488
```

Now we have to deal with the Linux LVM configuration. The Linux command-line tools for this want to operate on a disk device, not a disk image file. We can fake them out by using a virtual "loopback" device. The `losetup` command associates a loopback device with a raw disk image file.

We need to point the loopback device at the start of the LVM partition by specifying an offset *in bytes*. The `"$ ((...))"` syntax lets us do math on the command-line. Here we multiply the starting sector offset from the `mm1s` output by our 512 byte sector size (also shown in the `mm1s` output). We tell `losetup` to just grab the first available loopback device name (`"-f"`) and make the device read-only (`"-r"`). The read-only switch is actually redundant, since `ewfmount` only permits read-only access to the `ewf1` file. But it's good to develop careful habits.

But how do we know which loopback device `losetup` used? `"losetup -a"` displays all currently configured loopback devices and where they are pointing. Ours is the first loopback device, `/dev/loop0` (that's a zero not an oh).

The `file` command tells us that the loopback device is pointing to a Linux LVM v2 Physical Volume ("`LVM2 PV`"). So we are on the right track!

ACTIVATE LVM

```
# pvdisk /dev/loop0
--- Physical volume ---
PV Name                /dev/loop0
VG Name                VulnOSv2-vg
PV Size                31.26 GiB / not usable 0
...
# vgscan
Reading all physical volumes.  This may take a while...
Found volume group "RD" using metadata type lvm2
Found volume group "VulnOSv2-vg" using metadata type lvm2
# vgchange -a y VulnOSv2-vg
# lvscan | grep VulnOSv2-vg
ACTIVE                  '/dev/VulnOSv2-vg/root' [30.51 GiB] inherit
ACTIVE                  '/dev/VulnOSv2-vg/swap_1' [768.00 MiB] inherit
```

`pvdisk` gives more detail about the LVM physical volume. Of particular interest is the volume group's name— we're going to need this for later commands. In the example on the slide, the volume group name is "VulnOSv2-vg".

`vgscan` automatically scans disk and loopback devices for LVM metadata. The command finds the "RD" volume group from my local Linux analysis workstation as well as the "VulnOSv2-vg" volume group from our forensic image.

Activate an LVM volume group with "`vgchange -a y`". Activation assigns each of the different volumes within the LVM configuration to a Linux device node. We can see the various node names in the output of `lvscan`. By default, the device node path will always contain the volume group name.

The device nodes you see on the slide are the actual Linux file systems. If you wanted to acquire an image of the raw file system, then use `ewfacquire` or `dc3dd` on `/dev/VulnOSv2-vg/root`. But I'm more interested in mounting this file system so that I can find and extract artifacts with standard Linux command-line tools.

CHECK THE FILE SYSTEM

```
# fsstat /dev/VulnOSv2-vg/root
FILE SYSTEM INFORMATION
-----
File System Type: Ext4
Volume Name:
Volume ID: 46c34db340bee5aa35423fd055183259

Last Written at: 2019-10-05 05:41:50 (EDT)
Last Checked at: 2016-04-03 12:05:48 (EDT)

Last Mounted at: 2019-10-05 05:41:50 (EDT)
Unmounted properly
Last mounted on: /
...
```

Here I'm using `fsstat` to confirm the device nodes were set up properly. Looks like an EXT4 file system, last mounted as `/` (the root file system).

Note that the `fsstat` output says the file system was "Unmounted properly". So mounting it should be easy. Unfortunately, this turns out not to be the case, as we will see on the next slide.

DIRTY, DIRTY FILE SYSTEMS

```
# mkdir /mnt/test/data
# mount -o ro,noexec /dev/VulnOSv2-vg/root /mnt/test/data
mount: wrong fs type, bad option, bad superblock on /dev/mapper/Vuln...
       missing codepage or helper program, or other error
       In some cases useful info is found in syslog - try
       dmesg | tail or so

# dmesg | tail
[13458...] EXT4-fs (dm-6): INFO: recovery required on readonly file
system
[13458...] EXT4-fs (dm-6): write access will be enabled during recovery
[13458...] Buffer I/O error on device dm-6, logical block 0
[13458...] lost page write due to I/O error on dm-6
...
[13458...] JBD2: recovery failed
[13458...] EXT4-fs (dm-6): error loading journal
```

When I attempt to mount the file system, I use the “ro” (“read-only”) switch even though the loopback device and the underlying “ewf1” file are also set to read-only. Practice good forensic habits!

Another good habit when analyzing Linux disk images from Linux systems is to use the “noexec” flag, which is a software switch that prevents executing programs from the mounted disk image. You wouldn’t want to inadvertently run malware from the image you were investigating!

Unfortunately, the `mount` command fails. Digging into the matter with `dmesg`, it appears that the file system is underplayed (despite what `fsstat` told us). Note that the EXT4 driver is trying to make the file-system writable in order to clean up the file system— despite our “ro” option! Happily both the loopback device and `ewfmount` are blocking any changes, so our `mount` command just errors out.

As bad as this looks, there is a work-around which we can use to get the file system mounted. More on that on the next slide.

THE DIRTY SECRET

```
# mount -o ro,noexec,noload /dev/VulnOSv2-vg/root /mnt/test/data
# ls /mnt/test/data
bin    dev    home      lib        media     opt      root    sbin    sys    usr
boot  etc    initrd.img lost+found  mnt      proc    run     srv     tmp    var
#
# mount -o ro,noexec,loop,offset=$((2048*512))
# ls /mnt/test/data/boot
abi-3.13.0-24-generic      memtest86+.bin
config-3.13.0-24-generic  memtest86+.elf
grub                      memtest86+_multiboot.bin
initrd.img-3.13.0-24-generic System.map-3.13.0-24-generic
lost+found                vmlinuz-3.13.0-24-generic
```

The trick is to also use the “`noload`” option, which tells the file system driver to ignore any incomplete transactions in the file system journal. Usually the file system is in good enough shape to mount, even ignoring the unfinished changes in the journal.

The first `mount` command mounts the root file system on our target directory using the LVM device node name we set up earlier via the `vgchange` command. The `mount` command is silent if everything works, but we can use “`ls`” to get a directory listing of the top-level directory.

We can also mount the `/boot` partition directly. We need to set up a loopback device for this, but the `mount` command will accept “`loop`” and “`offset`” options and set up the loopback device for us. If you recall, `/boot` is an EXT2 file system, and EXT2 does not have a file system journal. So the “`noload`” option is not necessary here.

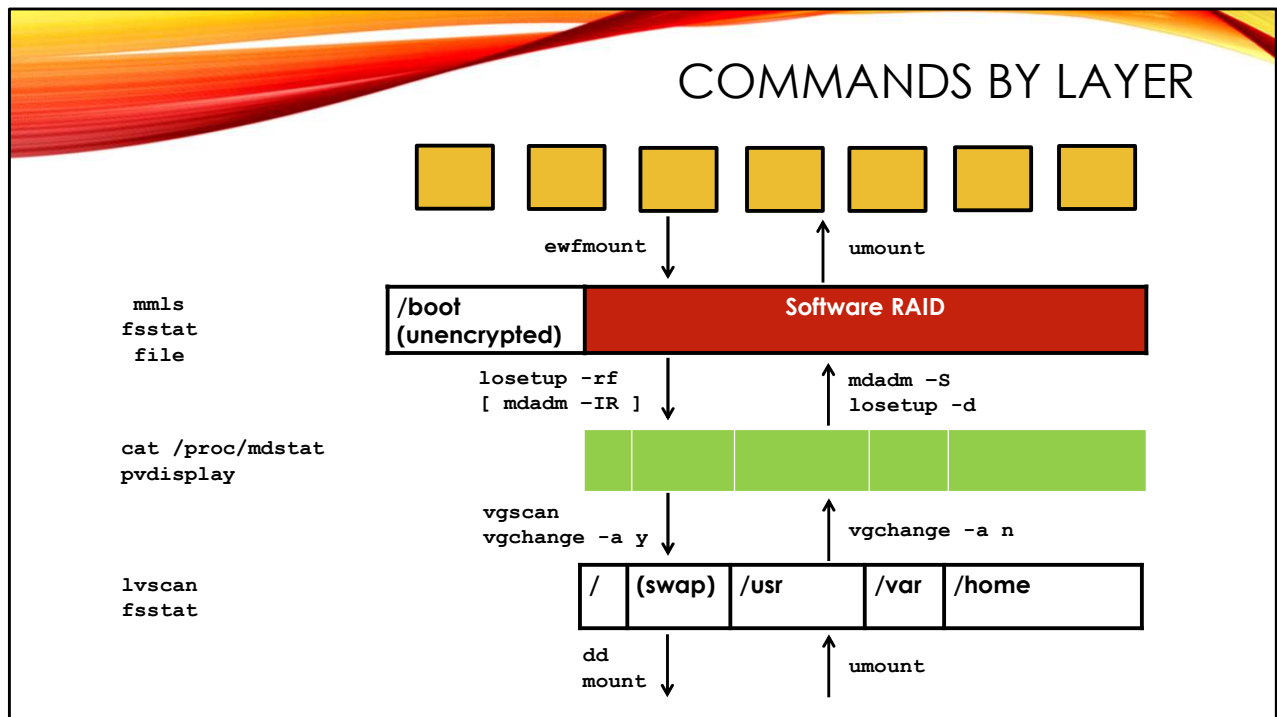
TEARDOWN

```
# umount /mnt/test/data/boot
# umount /mnt/test/data
#
# vgchange -a n VulnOSv2-vg
  0 logical volume(s) in volume group "VulnOSv2-vg" now active
#
# losetup -d /dev/loop0
#
# umount /mnt/test/img
```

Once you are done investigating, you will want to unmount and discard all of the various file systems and devices that you have created during this process.

Essentially we do everything in reverse order, using slightly modified commands:

1. Unmount any mounted file systems. We have to `umount .../boot` before the OS will let us `umount` the root file system that `/boot` is mounted on top of.
2. “`vgchange -a n`” to deactivate the `VulnOSv2-vg` volume group and discard the device nodes associated with the file systems.
3. “`losetup -d`” deletes our loopback device which was pointing at the beginning of the LVM2 volume.
4. Finally, we `umount` the virtual `ewf1` file that `ewfmount` created under `/mnt/test/image`



Here's a summary of the commands used to set up and tear down each layer of a Linux disk configuration. Note that I've included commands for interacting with a Linux software RAID configuration. You're going to get some practice with that in the lab exercise!

If you're looking for a similar chart for dealing with a disk image that includes a Linux encrypted volume, please see this presentation:

<http://deer-run.com/~hal/CEIC-dm-crypt-LVM2.pdf>

LAB – DISK MOUNTING

Let's try something a little more challenging...

Let's try this with multiple disks in a software RAID configuration!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Now that we have our file systems mounted. Let's do some quick triage and perhaps find some evil!

IMPORTANT DIRECTORIES

/etc	[%SystemRoot%/System32/config]
<i>Primary system configuration directory</i>	
<i>Separate configuration files/dirs for each app</i>	
/var/log	[Windows event logs]
<i>Security logs, application logs, etc</i>	
<i>Logs normally kept for about 4-5 weeks</i>	
/home/\$USER	[%USERPROFILE%]
/root	
<i>User data and user configuration information</i>	

There are potentially interesting artifacts all over the Linux file system, but the most important items tend to cluster in a few directories. Although things are not exactly the same, I'm also trying to give you the closest Windows equivalents to some of these directories.

`/etc` is where system and application configuration data tends to live. Applications will typically put their configuration files in directories under `/etc`. For example, `/etc/apache2` or `/etc/httpd` for the web server configuration.

Critical system logs live under `/var/log`. We will have a lot more to say about logs later on in this course.

User home directories are generally found under `/home`. The exception is that the home directory for the "root" (administrative) user is `/root`.

Also look out for what's happening in `/tmp` and `/var/tmp`. Exploits that do not gain system-level privileges will often write payloads into these directories. You'll be finding a lot of cryptocurrency miners running out of `/tmp`!

BASIC SYSTEM INFO

Linux distro name/version number:
`/etc/*-release`

Computer name:
`/etc/hostname`
Also log entries under `/var/log`

IP address(es):
`/etc/hosts` (static assignments)
`/var/lib/NetworkManager` (DHCP)
`/var/lib/dhclient` or `.../dhcp`

It's often important to know which version of Linux the system is running. Not only do some artifacts change location depending on the version of Linux, knowing the Linux version can also inform you as to which vulnerabilities your adversary might be exploiting. Linux systems generally have a file called `/etc/os-release` that contains version information. There may also be version-specific files like `/etc/redhat-release` on RedHat Enterprise Linux, Fedora, and CentOS and `/etc/lsb-release` on Debian and Ubuntu.

The system hostname is usually found in `/etc/hostname`. Standard Linux log messages also include the hostname. Older logs (possibly recovered from unallocated) might show if the system's name has been changed.

If the system uses a statically assigned IP address, it is usually found in the `/etc/hosts` file. DHCP lease information is typically found in `/var/lib/NetworkManager` on recent Linux systems, and `/var/lib/dhclient` or `/var/lib/dhcp` on older versions of Linux. Note that Linux systems will often keep a long history of historical DHCP lease information—possibly as far back as the initial system install! This is great for putting the system at a particular place at a particular time.

INSTALLATION DATE/TIME

Linux OS does not generally track installation date/time

Create time of `/lost+found` is good proxy for system install

Timestamps on SSH host keys typically indicates first boot
Key files are `/etc/ssh/ssh_host*_key`

Unlike Windows, which tracks system installation date/time in the registry, Linux systems generally do not save information regarding the system installation date. So we are left with using proxies to infer the installation date.

The “`lost+found`” directory at the top of each file system is created when the file system is made—generally during the system install. Linux file systems did not have creation dates until EXT4, but since the `lost+found` directory is generally untouched once it is created, the last modified (*mtime*) on the directory is usually sufficient. Here’s an example that uses the file systems we mounted in the last section:

```
# stat /mnt/test/data/lost+found
[...]
Access: 2016-04-03 16:05:48.000000000 +0000
Modify: 2016-04-03 16:05:48.000000000 +0000
Change: 2016-04-03 16:05:48.000000000 +0000
Birth: 2016-04-03 16:05:48.000000000 +0000
```

Note that on older Linux systems, the `stat` command may not show the `Birth` time. On these systems, use `debugfs` to view creation dates. `debugfs` labels this timestamp `crttime` for “creation time”.

The SSH host keys found under `/etc/ssh` are usually created the first time the system boots. So timestamps on these files are another way to assess the age of the system:

```
# stat /mnt/test/data/etc/ssh/ssh_host_rsa_key
[...]
Access: 2019-10-05 09:41:55.184710916 +0000
Modify: 2016-04-16 13:10:22.917943668 +0000
Change: 2016-04-16 13:10:22.917943668 +0000
  Birth: 2016-04-16 13:10:22.917943668 +0000
```

So we have a system image that was installed on April 3, 2016 but apparently not booted until April 16. This was a virtual machine image that may have been cloned and booted multiple times from a common baseline image.

Note that while Linux file systems store timestamps internally in UTC, Linux command-line programs default to displaying times in *whatever the default time zone for your analysis workstation might be*. But you can have commands display in whatever time zone you feel like by using the `TZ` environment variable:

```
# date
Wed Feb 26 13:33:17 EST 2020
# ls -l /etc/passwd
-rw-r--r-- 1 root root 2095 Jan 29 16:12 /etc/passwd
# export TZ=UTC
# date
Wed Feb 26 18:33:34 UTC 2020
# ls -l /etc/passwd
-rw-r--r-- 1 root root 2095 Jan 29 21:12 /etc/passwd
```

DEFAULT TIME ZONE

System logs written in default time zone for machine

`/etc/localtime` stores default time zone data

Binary file format:

- Use "zdump" on Linux

- "strings -a `/etc/localtime`" often works

- Look for matching file under `/usr/share/zoneinfo`

And speaking of time zones, it is important you know the default time zone for the system you are investigating. Linux log files and other important artifacts contain timestamps written in the local time zone for the machine.

The system default time zone is stored in the `/etc/localtime` file. This file is in a binary format. While running "strings" on the file will often give you clues, the easiest thing to do if you are running from a Linux analysis host is to use the `zdump` command:

```
# zdump /mnt/test/data/etc/localtime
/mnt/test/data/etc/localtime  Wed Feb 26 19:40:14 2020  CET
```

It looks like our sample image was set to Central European Time (CET).



POST-EXPLOITATION GOALS

Back doors

Persistent malware

Now that we have a good idea of the basic configuration of the system, let's go hunting for evil.

In general, attackers will want some sort of back-door access into the compromised system and a way for their malware to be started automatically after the system boots. Note that neither one of these is necessarily a given—I've seen cryptocurrency miners dropped onto systems opportunistically with no particular care given to persistence. I supposed the attackers feel that they could just re-compromise the system and drop another miner.

COMMON BACK DOORS

Custom malware installs

- New or replacement binaries
- Web shells

Account modification

- New (admin) accounts added
- Application role accounts unlocked
- Enhanced “sudo” access privileges
- `$HOME/.ssh/authorized_keys` entries added

Back doors could take the form of custom malware implants. A web shell is often the easiest route, particularly if the attacker is exploiting a web app vulnerability to gain access. Another common back door in the Linux universe is a replacement SSH service with a hard-coded username/password for gaining admin access.

Another back-door approach is leveraging existing accounts— particularly application accounts like *www* or *mysql* that are normally locked. If the attacker sets a re-usable password on these accounts, they could use them to access the system remotely. Creating an `authorized_keys` entry in the user’s home directory is another way of opening up access to the account.

Any account with user ID zero has admin-level access. Normally there should be only a single “root” account with UID 0 in the password file, but multiple UID 0 accounts are allowed. “`sort -t : -k 3 -n /etc/passwd`” will sort the `passwd` file numerically by UID, so it will be easy to see UID 0 accounts, even if your attacker adds them in the middle of the file.

Note that the `sudo` command also gives admin privileges. Look for modifications to `/etc/sudoers` or groups this file refers to in `/etc/group` such as “admin” or “wheel” group entries.

PERSISTING MALWARE

Service start-up scripts

`/etc/systemd/system,`
`/usr/lib/systemd/system`
`/etc/init*`

(systemd)

(traditional and Upstart)

Scheduled tasks

`/etc/cron*`
`/var/spool/cron/crontabs`
`/var/spool/cron/atjobs`

Attackers may use the normal service start-up mechanisms to restart their malware. On modern Linux systems that use Systemd, service startup configuration is found under `/usr/lib/systemd/system` and `/etc/systemd/system`. Older systems use configuration files under directories named `/etc/init*`.

Look for recent changes to files under these directories. Note that in some cases these files may invoke other scripts that might have been modified by the attacker. This is much less obvious than the attacker modifying the start-up configuration files themselves.

Scheduled tasks can also be used to start persistent malware. There are multiple places to look because Linux systems operate multiple task-scheduling systems in parallel. Again, attackers may modify scripts invoked by legitimate scheduled tasks rather than creating or modifying the scheduled tasks directly.

RECENT MODIFICATIONS

```
find /mnt/test/data -newermt '2023-07-24 00:00:00'
```

Display files modified after a certain timestamp

```
find /mnt/test/data -newer /mnt/test/data/etc/passwd
```

Display files modified after target file

```
find /mnt/test/data -mtime -7
```

Find files modified in the last week

```
ls -lArt /mnt/test/data/etc
```

Directory listing sorted by mtime, oldest first

So it's a good idea to look for any recent modifications to the system. Yes, an attacker with admin-level access can reset file timestamps, but it's amazing how often they don't bother.

The `find` command lets you search your mounted image based on different criteria, including timestamps. If you know the date/time attackers were active on the system, you can use the `-newermt` option to see files that have been modified since a specific date and time. Or if you find a file that has been modified by the attacker, you can use the `-newer` option to see other files that were modified after the attacker changed a given target file. Or if the changes were recent, you could just ask for all files modified ("`-mtime`") less than seven days ("`-7`", "more than seven" would be "`+7`") ago.

It is often convenient to list a directory in order of last modification rather than alphabetically. "`ls -rt`" is a reverse sort by time (oldest to newest), "`-l`" gives file details (that's an `el` for "long listing"), and "`-A`" shows "hidden" files whose name starts with a period.

LAB – DISK TRIAGE

How quickly can you find evil?

Get some practice profiling systems and quickly finding artifacts of compromise.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



TIMELINE ANALYSIS

Timeline analysis is a fast way to find intrusion artifacts during an investigation.



ALL HAIL TIMELINE ANALYSIS!

Attackers leave breadcrumbs all over:

- Program installation and execution
- File modification
- User account usage

A *timeline* puts the breadcrumbs in chronological order

- Helps tell the story of your compromise
- Directs you to important evidence

Many attacker activities during an intrusion leave tracks behind in the file system. For example:

- An exploit may drop a web shell onto your system. The creation date on the file containing the web shell helps date the start of the incident.
- The attacker may then use the web shell to download additional malware, which will have its own set of timestamps.
- Next the attacker succeeds at privilege escalation and suddenly root-owned files on the system begin being updated.
- The attacker modifies configuration files, leaves behind back-doors, etc.

A timeline shows you these changes in chronological order and helps tell the story of what happened. It directs you to files that were modified or added by the attacker that you may have not seen yet.

STANDARD TIMESTAMPS

Last modified time (M)

Last time the file contents were changed

Last access time (A)

Last time the file was viewed/executed*

Metadata change time (C)

Last inode update (`chown`, `chmod`, ...)

Creation time (B)

Date/time of file creation (EXT4 only)

Timestamps are created using the four standard file timestamp types:

Last modified (mtime) – The last time the content of the file was changed. For example, when a new file is created or you use an editor to make changes to a file.

Last access time (atime) – The last time the contents of a file have been read. If the file is a program or script, atime usually represents the last time the program was executed. However, Linux systems generally do not update atimes every time the file is read, as we will discuss below.

Metadata change time (ctime) – The last time metadata about the file is updated. For example, changing the file owner with `chown` or the file permissions with `chmod`.

Creation time (btime) – The date the file was created. Creation time is generally referred to as the “btime” (born-on date) to distinguish it from the metadata change time (ctime). However, some Linux commands (like `debugfs`) refer to this timestamp as “ctime”. btime was only added to EXT file systems with EXT4 (it is also found in modern versions of XFS).

File system developers have realized that updating atime on every single file access is inefficient, because it means you have to write the update into the file system even when the file is just being read (or executed) over and over again. Windows NTFS stopped updating atimes back in Windows 7.

Linux systems typically use a file system option known as “relatime”. With this option, atimes are updated on file access if either:

1. The atime is older than the mtime or ctime (hence a “relative atime update” or “relatime”)—this is designed for programs like mail readers that want to know if the file has been accessed since it was last updated.
2. The atime was last updated more than 24 hours ago.

So atimes in Linux are only updated on an occasional basis, but are still sometimes useful. For example, atime updates on programs that are not commonly used or on malware dropped by the attacker can still be important artifacts of execution in your timeline. The atime will generally be updated the *first* time the program is executed in a given 24-hour window.

TIMELINE CAVEATS

Timestamps are ephemeral

- You only get the *last* modified time, change time, etc

- Normal system usage will update timestamps

- Admin users may change timestamps at will

Analyst needs understanding of typical system behaviors

Timelines are a guide to evidence, not evidence themselves

It's important to understand the limitations of timelines. Remember that you only get the *last* modified or access time on a file. It's possible that the attacker modifies `/etc/shadow` to set a password on an account like the "postgres" database user. But then a regular user might come along and change their password, updating the mtime on the file. You've lost a potentially useful piece of information—when the attacker updated `/etc/shadow`—and you now have a "hole" or "gap" in your timeline.

Also, timestamps on files can be updated arbitrarily by the superuser. The `touch` command allows root to set the atime or mtime to any time desired. `debugfs` gives the ability to update any timestamp (for examples see <http://blog.commandlinekungfu.com/2010/02/episode-80-time-bandits.html>).

Even without attacker anti-forensics, reading a timeline requires an understanding of typical system behaviors. You see an atime update on `debugfs` that was probably due to enemy activity, but what does that command do for the attacker in the context of the incident? What does it mean if the "set-UID" bit is turned on for an executable?

So it takes an experienced technical analyst to understand what the timeline is saying. It's unlikely that you'll be using your timeline as direct evidence. But it's a great guide to help you find evidence!



HOW TO TIMELINE

1. Collect raw data into a *body file*
2. Create chronological output (usually as CSV)
3. Jump to key *pivot points* for analysis

To create a timeline you first need to extract the raw timestamp data into a file. These files are often referred to as *body files*. The name comes from an early Open Source forensic toolkit called the Coroner's Toolkit (TCT). TCT contained a program called `graverobber` for extracting timeline information. And what do grave robbers steal? They steal *bodies* of course! The name *body file* has stuck even though we don't use `graverobber` anymore.

Once you have your body file(s), we need a tool to create the sorted timeline. Timelines are often created as CSV files, which are easier to search, filter, and annotate. Some analysts use MS Excel to read their timelines, but a better option is Eric Zimmerman's free Timeline Explorer tool (TLE). TLE is much faster, especially with large timelines, and has powerful sorting, filtering, and tagging capabilities. For all of Eric's great tools, visit <https://ericzimmerman.github.io/>

But where do you start looking? Hopefully your earlier triage will give you some places to start. For example, in a previous lab exercise we investigated attacker changes to the `/etc/passwd` and `/etc/shadow` files. So jump to the last `mtime` update on these files and look at what else was happening around that same time. Or look for the creation time of malware the attacker might have left behind. We call these kinds of markers *pivot points*—they are the starting points for your analysis.

STEP 1 – COLLECT DATA

```
# mkdir /cases/timeline
# cd /cases/timeline
#
#
# fls -r -m / /dev/mapper/VulnOSv2--vg-root | gzip >bodyfile-root.gz
#
# fls -o 2048 -r -m /boot /mnt/test/img/ewfl | gzip >bodyfile-boot.gz
```

Recursive – process all files/directories

mactime format & mount prefix

Sector offset (from mmls)

Recognized file system type

Body files are quickly generated with a Sleuthkit tool call `fls`. Standard arguments include:

- “-r” to *recursively* read through the entire file system (rather than just dumping information from the top-level directory, which is the default). You want to be sure to collect evidence from all files and directories.
- “-m <mntpt>” to specify the output format of `fls` should be in *mactime* format (which is simply a pipe-delimited text file). We will be using *mactime* in the next step to make our timeline. The <mntpt> argument to -m is the path the file system is normally mounted on—see the second example on the slide where we are dumping data from `/boot`. The mount pathname will be added to the front of the file paths in the `fls` output so that the path names are consistent with the way the file system was used on the live machine.
- “-o” lets you specify a sector offset into a full disk image to find the start of the file system

You must also specify a raw file system of a type TSK tools can recognize. The EXT4 `/boot` file system can be accessed directly from the raw disk image created by `ewfmount` (and if TSK is compiled with `libewf` support, it can read the E01 files directly). But TSK doesn't understand Linux LVM, so we must first associate the logical volumes with disk devices that `fls` can read.

Note that because `mactime` format body files are just plain ASCII text, they compress very well. So we are `gzip`-ing them to save space.

While some analysts will concatenate all of their body file data into a single large file, I prefer to dump each file system as a separate body file. That way, if I mess up one command, I only have to rebuild that one body file. Otherwise the bad data from my one wrong command might pollute the file with all of my other good data.

STEP 2 – BUILD TIMELINE

```
# zcat bodyfile-* |  
  mactime -d -y  
    -p /mnt/test/data/etc/passwd  
    -g /mnt/test/data/etc/group  
2019-10-01 >timeline.csv
```

*-d for CSV (delimited) output
-y for ANSI dates in UTC*

*Specify location of
passwd/group files
to see names not
UIDs and GIDs*

*Build timeline from
this date onwards*

*Save output
to file*

Once we have all of our body file data collected, we feed it into the `mactime` tool to produce our timeline. Here I'm using `zcat` to uncompress the body files I made in the previous step and piping the uncompressed output into `mactime`.

Useful arguments to `mactime` include:

- “-d” to produce *delimited* (CSV) output
- “-y” for ISO 8601 date output in UTC (2019-10-05T11:31:37Z)
- “-p” and “-g” to specify the location of the `passwd` and shadow files from the image you are analyzing so you see the right user and group names in the output

You may optionally specify a single date as we are doing in the example on the slide or a date range (2019-10-01..2019-11-01). Single date means create the timeline using only timestamps from that date forwards. Range of dates means only output times within the dates specified.

Output normally goes to the standard output (the terminal). Here we are using output redirection to save the output in a file.



STEP 3 – ANALYZE!

Questions to answer:

How/when did the attacker breach the system?
How/when did they gain root access?

What are you looking for?

Suspicious file/directory creation or modification
Evidence of program installation and/or execution

Use your pivot points to begin your analysis

Once you have your timeline, the rest of the work is analysis. Ultimately, intrusion analysis tries to answer at least two important questions— how did they break in and how did they get admin privileges? “What did they take?” is another question that is often asked. The kind of evidence you can see in the timeline is changes to the file system— attackers adding files or directories, modifying or replacing existing files, making permissions changes, etc.

To find the evidence, think about possible pivot points in the timeline based on what you already know from your triage:

- If the attacker is running custom malware, look for the btime of the malicious executable and possibly its installation directory.
- Our attacker added accounts, modifying `/etc/passwd` and `/etc/shadow`— start at the mtime updates on these files.
- Maybe you have an IDS alert or information from your logs that indicate attacker activity. Jump to these times in your timeline and see what was happening in the file system.

LAB – TIMELINE ANALYSIS

Jump in and swim!

The best way to learn timeline analysis is to try it yourself... with a little expert guidance!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



CORE LOG ANALYSIS

Logs are an essential part of the forensic analysis of any operating system.



LINUX LOGS

Generally found under `/var/log`

Logs are primarily text
Easy to modify and manipulate

Logging is discretionary
Amount and format of logs left to developers

Linux logs are generally found under `/var/log`. This is largely convention, however– they could be written anywhere in the file system and you will find them other places on other Unix-like operating systems.

Unix logs are usually simple text files. It is very easy for attackers who have obtained admin access to edit or simply remove log files. Attackers have even created tools to modify the common binary log formats on Linux which we will be discussing shortly. So it is a good idea to ship copies of your system to some other protected storage area.

Linux logging is *discretionary*– the software developers decide what they are going to log and the format they are going to log it in. This can make automated log analysis frustrating, because the logs are so free-form. And of course attacker tools are not going to provide helpful logging information because they don't have to.

LAST LOGIN HISTORY

wtmp – User logins and system reboots [read with **last**]

File may be truncated weekly or monthly

btmp – Failed logins [read with **lastb**]

Often not kept due to risk of password disclosure

lastlog – Last login for each user [read with **lastlog**]

Varying formats make decoding tricky

The `/var/log/wtmp` file stores a record of login sessions and reboots. It is in a special binary format, so you have to use the `last` command to dump out information:

```
# last -if /mnt/test/data/var/log/wtmp
mail pts/1 192.168.210.131 Sat Oct 5 07:23 - 07:24 (00:00)
mail pts/1 192.168.210.131 Sat Oct 5 07:21 - 07:21 (00:00)
mail pts/1 192.168.210.131 Sat Oct 5 07:18 - 07:19 (00:00)
mail pts/1 192.168.210.131 Sat Oct 5 07:13 - 07:18 (00:04)
reboot system boot 0.0.0.0 Sat Oct 5 05:41 - 13:42 (155+...)
root tty1 0.0.0.0 Wed May 4 13:36 - down (00:01)
vulnosad pts/0 192.168.56.101 Wed May 4 13:35 - 13:36 (00:00)
root tty1 0.0.0.0 Wed May 4 13:34 - 13:34 (00:00)
reboot system boot 0.0.0.0 Wed May 4 13:33 - 13:37 (00:03)
root pts/0 192.168.56.101 Wed May 4 13:01 - down (00:06)
vulnosad pts/0 192.168.56.101 Wed May 4 12:57 - 13:00 (00:03)
reboot system boot 0.0.0.0 Wed May 4 12:56 - 13:07 (00:10)
[...]
```

The “`-i`” flag shows IP addresses rather than hostnames, and “`-f`” allows you to specify a file path that is not the default `/var/log/wtmp` file. `last` shows the newest logins first.

The first part of the output shows remote logins by the “mail” account from IP address 192.168.210.131. Then we see a system reboot in the log. The next line is a login by “root” on the local text-mode console of the system– “tty1” (if the login had occurred on the graphical console you would see “:0” in the IP address column).

The `btmp` file stores information about failed logins, but it does not exist by default. Many administrators choose not to enable `btmp` logging because it can sometimes disclose user passwords– how many times have you accidentally typed your password into the username field? If you have a `btmp` file, you can read it with the `lastb` command:

```
# lastb -if /mnt/test/data/var/log/btmp
mail  ssh:notty  192.168.210.131  Sat Oct  5 07:20 - 07:20 (00:00)
root  ssh:notty  192.168.210.131  Sat Oct  5 06:52 - 06:52 (00:00)
root  ssh:notty  192.168.210.131  Sat Oct  5 06:52 - 06:52 (00:00)
root  ssh:notty  192.168.210.131  Sat Oct  5 06:52 - 06:52 (00:00)
root  ssh:notty  192.168.210.131  Sat Oct  5 06:52 - 06:52 (00:00)
[...]
```

We can see a failed login for user “mail” and multiple failed “root” logins, all originating from IP address 192.168.210.131.

The `lastlog` file stores last login information for each user on the system. The file can appear to be huge, but it is actually a sparse file– the offset to any user record is their UID times the size of the `lastlog` record. You read the file with the `lastlog` command, which simply goes line by line through the password file and dumps the `lastlog` record for each UID it finds there. That means if you are not using the password file from the system the `/var/log/lastlog` file was taken from, or if you are but there were existing user accounts that have been deleted from that password file, then you may not be seeing all the data in the file.

The biggest problem, however, is that the format of the `lastlog` file is highly variable. The version of Linux you are running as well as the processor architecture that the `lastlog` file was written on can affect the size of the `lastlog` records and impact your ability to read the file. Stefan Johnson has written a generic `lastlog` parsing tool in Python that can help:

https://github.com/tigerphoenixdragon/lastlog_parser



SYSLOG

Syslog is the background service that receives/routes logs

Destination is usually local log files

Default is restart logs weekly, keep four previous weeks

Can also route logs to other hosts over the network

Always a good idea to aggregate longer term log history

The primary logging service on Linux systems is Syslog. It runs as a background service and receives log messages from various processes on the system (and the OS kernel) and then routes the logs messages to different destinations.

Typically, log messages are stored in text files on the local system. There is an external “log rotation” process that is responsible for making sure the log files don’t grow forever and fill up the disk. Log rotation usually happens weekly— the old log file is renamed and sometimes compressed, and a new log file is started. Traditionally, Linux systems will keep four weeks of old log files in addition to the log file that is currently being written to by Syslog. So you’ll find about a month worth of logs under `/var/log`. If `/var/log/secure` is the primary file name, you’ll find the older logs in files named `secure.1` through `secure.4`, with `secure.4` holding the oldest log messages.

However, there is also a Syslog network protocol that allows routing logs over the network to a Syslog service on another host. This is useful for aggregating your logs together into a SEIM tool or other log analysis platform—collected logs have huge value during an investigation. Having a copy of your logs on a different system also helps protect them from attackers destroying the logs on the local machine.

SYSLOG CONFIGURATION

Type of log messages by "facility" and "priority"	Local file destinations
auth,authpriv.*	/var/log/auth.log
.;auth,authpriv.none	-/var/log/syslog
#cron.*	/var/log/cron.log
#daemon.*	-/var/log/daemon.log
kern.*	-/var/log/kern.log
#lpr.*	-/var/log/lpr.log
mail.*	-/var/log/mail.log
auth,authpriv.*	@loghost
*.notice;auth,authpriv.none	@loghost
	Send logs to remote host

Here is part of a typical Syslog configuration file (look for the config files as `/etc/rsyslog*` or `/etc/syslog-ng*`, or `/etc/syslog.conf` on older Unix systems). The left column describes what the administrator wants to log and the right column is the destination where the log messages should be sent.

The left column uses a combination of "*facility.priority*" to select log messages. The *facility* tells something about where the log message came from. For example, "authpriv" messages are authentication or security messages that are supposed to be kept private (for administrators only). Messages like these are used to track user logins, logouts, and privilege escalations and therefore are very interesting to us. *Priority* ranges from debug (lowest) all the way up to emergency (highest). The "*" is a wildcard that means match any facility or priority.

The destination for the log messages can be a file path or a remote hostname (or IP address) given as "@<hostname>". When writing to a log file, Syslog normally tries to flush the log messages immediately to disk. The "-" sign in front of a log file name means that the logs are less critical and can be buffered before writing to disk. This is much more efficient in terms of file system performance.

SAMPLE LOG MESSAGES

```
Oct  5 13:13:53 VulnOSv2 sshd[2624]: Accepted password for mail from
192.168.210.131 port 57686 ssh2
Oct  5 13:13:53 VulnOSv2 sshd[2624]: pam_unix(sshd:session): session
opened for user mail by (uid=0)
Oct  5 13:14:04 VulnOSv2 sudo:      mail : TTY=pts/1 ; PWD=/var/mail ;
USER=root ; COMMAND=/bin/su -
Oct  5 13:14:04 VulnOSv2 sudo: pam_unix(sudo:session): session opened
for user root by mail(uid=0)
Oct  5 13:14:04 VulnOSv2 su[2721]: pam_unix(su:session): session
opened for user root by mail(uid=0)
Oct  5 13:18:48 VulnOSv2 sshd[2624]: pam_unix(sshd:session): session
closed for user mail
```

Timestamp in local time zone

Host where log originated

Process [and PID]

Here are some typical log messages from a Linux log file. Each log message starts with a date/time stamp *in the system's default time zone*, the name of the host where the log message was generated, and the process name and usually the process ID number of the software that generated the log. The rest of the log message is left up to the developers, and you can see that the format of the log messages varies widely.

The first two lines of log messages above show user “mail” logging in via SSH from IP address 192.168.210.131 (port 57686 is the source port on the remote system) at 13:13:53 on Oct 5. The next three lines show user mail doing “`sudo /bin/su -`”, which will give them an administrative shell. The last line shows the SSH session closing at 13:18:48 (use the process ID of the SSH process to match the session openings and closings in a busy log file).

Notice that the standard date/time stamps do not include the year. I’m assuming that the original Unix developers believed you wouldn’t keep log messages around longer than a month, so the year was unimportant. But if you do keep logs for a long time (or if you are recovering old log messages from unallocated), figuring out the year data becomes a factor. Sometimes you will see log messages (particularly kernel logs during the system boot) which contain the year in the text of the log message.

The format of the date/time stamp is very regular and can be searched for. This is a useful trick when trying to recover deleted log messages from unallocated. You can use the Unix regular expression '[A-Z][a-z]* *[0-9]* *[0-9]*:[0-9]*:[0-9]* *' to search for the standard date/time stamp (uppercase letter, lowercase letters, spaces, number, spaces, number, colon, number, colon, number, spaces).

Another overlooked part of the log messages is the host name. I've had cases where suspects changed the host name of their machine. I was able to determine the old host name for the system from older log messages.

Notice that the Syslog facility and priority of each log message is *not* logged. This information is only associated with the log message while it is being transmitted.

USEFUL LOGS

auth,authpriv.* – *All things security-related*

kern.* – *USB and other device info, firewall logs*

cron.* – *Scheduled task execution*

daemon.* – *Other applications and services*

Important security messages go to authpriv (auth on older Unix systems). Look for these messages first.

kern messages will contain information about devices on the system, including USB devices as they are plugged in (for more details about USB forensics in Linux see <http://blog.commandlinekungfu.com/2010/01/episode-77-usb-history.html>).

kern messages also can contain logs from the Linux Netfilter firewall, aka IP Tables:

```
Mar  3 20:54:10 caribou kernel: [559355.051255] [UFW
BLOCK] IN=eth0 OUT=
MAC=44:8a:5b:6d:c6:f2:34:db:9c:67:00:8a:08:00
SRC=35.241.56.184 DST=192.168.1.13 LEN=40 TOS=0x00
PREC=0x00 TTL=106 ID=0 DF PROTO=TCP SPT=443 DPT=37871
WINDOW=0 RES=0x00 RST URGP=0
```

These logs are dense and difficult to read, but they are very regular and can easily be parsed into a more readable format. A little Google-ing will turn up many tools that can parse these logs.

Scheduled task logs (cron.*) and logs from various system services (daemon.*) can also be useful.

LAB – LOG ANALYSIS

It may not be glamorous, but...

There is all sorts of useful information waiting for you in your logs!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



ADDITIONAL LOGS

Syslog style logs and `wtmp/btmp/lastlog` are common, but there are other types of logs you may run into on Linux systems that can be very useful in investigations.



OTHER USEFUL LOGS

Web server logs

Often document the initial compromise

Kernel audit logs

Optional mandatory logging, very detailed

Other application logs

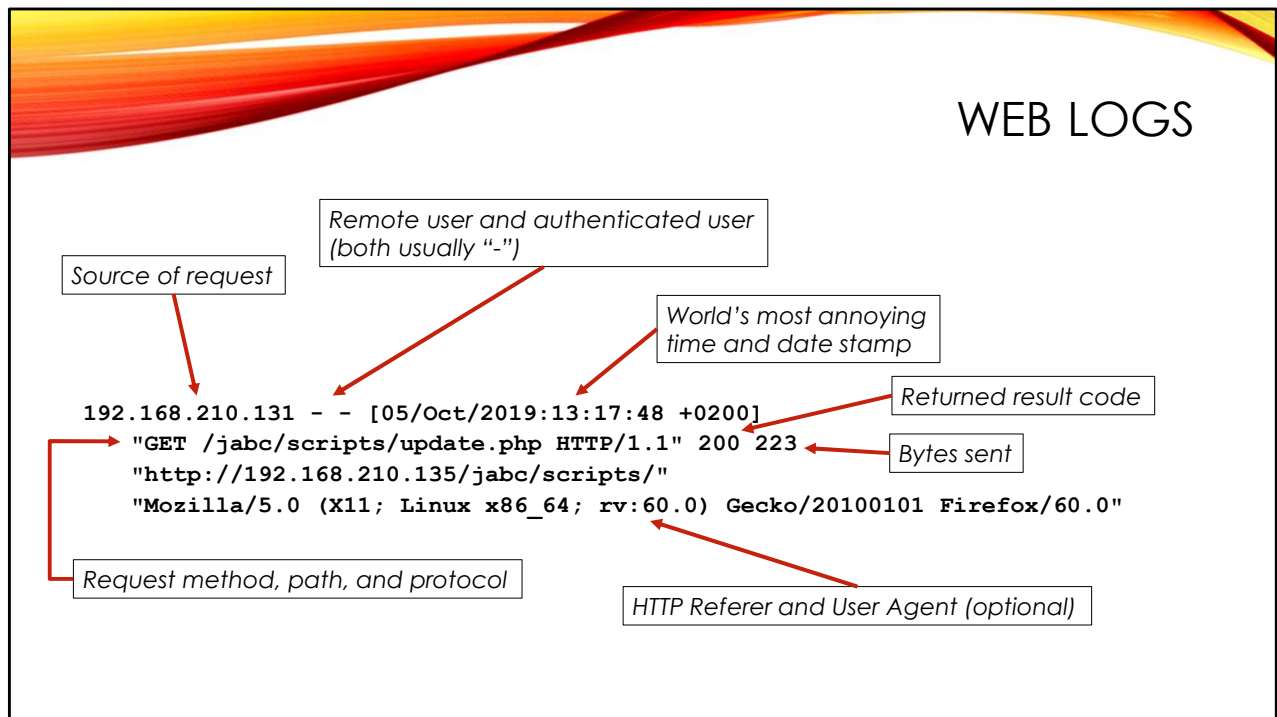
Databases, web proxies, ...

Given the number of web server exploits, you will be spending a lot of your life looking at web server logs. Your web server logs can document when the breach occurred and where the attackers originated from. They may also possibly supply some details about the nature of the exploit.

Linux systems may have kernel-level auditing enabled. This is similar to Windows Sysmon. The information is incredibly detailed but can be difficult to understand. Plus it needs specialized configuration in order to provide the most useful information. If you are the administrator of a Linux system, you might want to look into enabling this logging.

Although we won't have time to cover them here, logs from other services like databases and proxy servers can also be useful. Proxy servers tend to write logs by default. Database logging often needs to be increased to be useful—for example logging individual database queries is not normally enabled by default, but is incredibly useful after an incident. Even proxy logs can be enhanced by adding information such as browser user-agent and query string information.

WEB LOGS



Linux web servers—whether Apache or Nginx or something else—tend to use a standard log format developed for the NCSA server in the early days of the web. There are a lot of things I don't like about this log format, but it's what we get by default. Happily there are a lot of tools that can parse these logs and do useful things with them.

Web logs are typically found in directories under `/var/log` like `/var/log/httpd` or `.../apache*` or `.../nginx`.

Here's a breakdown of the fields:

- IP address or hostname of the remote host— If you are in control of the web server, try to turn off DNS lookups and always log the raw IP address.
- Remote user and authenticated user— The remote user was supposed to be determined using the old "ident" protocol, which nobody supports anymore. The authenticated user is only known if the user is using HTTP Basic or Digest auth or some other built-in authentication strategy in your web server (hint: this never happens in modern web apps). So these fields are almost always "-", indicating no information.

- Date/time stamp— This has got to be one of the most unhelpful date/time stamp formats ever. It's not sortable—day of month first and month abbreviations instead of numbers? Who does that? It's written in system local time, but at least the time zone offset is provided (+0200 here means two hours ahead of UTC).
- Request method, URI path, protocol version
- Result code— 200 is success, 3xx is a redirect to another URL, 4xx is client error (like "404 Not found"—the client asked for something that didn't exist), 5xx is server error (can sometimes indicate an exploit that causes the server to blow up).
- Bytes sent— Note that this is *bytes sent* not the actual size of the requested object. For example, a large file transfer may have been interrupted in the middle and the client is coming back to get the rest of the object they are missing.
- HTTP Referer (optional)— HTTP referer is the web page that contained the link we clicked on to get to this web page. In the case of embedded elements like images, style sheets, and javascript, the referer is the web page those elements are used in. HTTP referer information may not be present in the default log format, but if it's your web server, you should definitely make sure referer logging is enabled.
- User-agent string (optional)— The user-agent string from the software making the web request. Useful for tracking malware that uses unique user-agent strings. Like referer, this field is optional and may need to be enabled in your logs.

There is one more important thing to note about timestamps in web logs. The timestamp is set at the time of the web request, but the log message is only put into the log file when the web server finishes processing the request. That means that it is possible to see web log messages with timestamps out of chronological order when you have web requests that take a long time to complete.

DON'T FORGET ERROR LOGS!

```
[...]
PHP Notice:  Use of undefined constant
aygiTmxlbIIsICRsZW4pOyAkbgVuID0gJGFbJ2xlbiddOyAkYiA9ICcnOyB3aGlsZS
Aoc3RybGVuKCRiKSA8ICRsZW4pIHsgc3dpdGNoICgkc190eXBlKSB7IGNhc2UgJ3N0
cmVhbSc6ICRiIC49IGZyZWFKKCRzLCAkbGVuLXN0cmxlbWVudG91Y291Y291Y291
FzZSAnc29ja2V0Jz0gJGIGLj0gc29ja2V0X3JlYWQoJHMsICRsZW4tc3RybGVuKC...
[Sat Oct 05 13:17:48.483593 2019] [:error] [pid 1789]
[client 192.168.210.131:41888] PHP Warning:  system():
Cannot execute a blank command in
/var/www/html/jabc/scripts/update.php on line 2,
referer: http://192.168.210.135/jabc/scripts/
[...]
```

In the first log message on the slide you see some base64 encoded exploit code. The second line shows the attackers trying to exploit the `update.php` script.

LINUX KERNEL AUDITING

Kernel-level activity monitor can see everything

- System booting

- User logins and privilege change/escalation

- Scheduled task execution

- SELINUX security policy violations

With additional configuration can log

- File access, modification, execution

- Any specific system call(s) across all processes

- User keystrokes

- Locally defined tags or keywords for later searching

Linux kernel auditing is an optional type of logging that may be enabled on some servers. I recommend enabling it on servers you control. If enabled, you will find the audit logs under `/var/log/audit` by default. Kernel auditing is mandatory— it happens independent of the application and is not left up to the software developer, but instead is configured by the system admin.

With no special configuration, kernel auditing will log user login/logout activity and privilege escalations, as well as scheduled tasks taking on various user roles. If you are running SELinux (even in “passive” or non-blocking mode), the SELinux logs end up in your audit logs as well. Note that attackers may forget to edit user login history in the audit logs when they are trashing your Syslog-style logs—comparing the two logs is sometimes enlightening.

However, you can also enhance logging levels to log file access, process execution, track specific system calls (the “`ausyscall --dump`” command will give you a list of system calls you can trace), and even perform keystroke logging (look for documentation on the `pam_tty_audit` module). Sample configurations can be found in the CIS Benchmark Guide for Red Hat systems ([cisecurity.org](https://www.cisecurity.org)) and <https://github.com/bfuzzy/auditd-attack>

Another useful note is that you can add your own keywords for individual rules in your audit configuration. A good set of unique keywords can make searching your audit logs much easier during an incident or a hunt.

ALL HAIL AUSEARCH!

```
# ausearch -if /mnt/evidence/var/log/audit -c useradd
----
time->Thu Feb 20 13:26:44 2020
type=PROCTITLE msg=audit(1582223204.906:342):
  proctitle=2F7573722F7362696E2F75736572616464002D64002F7573722F706870002D6D0
  02D2D73797374656D002D2D7368656C6C002F62696E2F62617368002D2D736B656C002F6574
  632F736B656C002D4700776865656C00706870
type=PATH msg=audit(1582223204.906:342): item=0 name="/etc/passwd"
  inode=135568 dev=fd:00 mode=0100644 ouid=0 ogid=0 rdev=00:00
  obj=system_u:object_r:passwd_file_t:s0 objtype=NORMAL
  cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=CWD msg=audit(1582223204.906:342): cwd="/var/mail"
type=SYSCALL msg=audit(1582223204.906:342): arch=c000003e syscall=2
  success=yes exit=5 a0=55d79f171ce0 a1=20902 a2=0 a3=8 items=1 ppid=9425
  pid=9428 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0
  tty=pts1 ses=3 comm="useradd" exe="/usr/sbin/useradd"
  subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="auth-files"
```

Audit logs are just raw text files, but the best way to search them is with the `ausearch` command. This is because `ausearch` converts the Unix epoch style timestamps in the audit log messages into a human-readable timestamp (you see the raw epoch timestamp in each log message too “...msg=audit (<epoch>:<auditID>) ...”). Also `ausearch` shows you all messages associated with the events you are looking for—like in the example on the slide where there are multiple audit associated with a single `useradd` command.

The “type=PROCTITLE” message contains a hex-encoded copy of the user’s command line. You can read it out as follows:

```
$ echo 2F7573722F736269... | xxd -r -p | tr '\000' ' '; echo
/usr/sbin/useradd -d /usr/php -m --system --shell /bin/bash -
-skel /etc/skel -G wheel php
```

Use `echo` to pipe the hex encoded data into `xxd`, which will convert it back into ASCII character data. The various command line arguments are null-delimited, so use `tr` to convert the nulls to spaces. The last `echo` command adds a newline to the end so you can read the command-line more easily.

The “type=PATH” message shows the file the `useradd` command is changing. Here it’s the `/etc/passwd` file, but there’s another similar set of audit messages showing changes to `/etc/shadow` and so on.

The “type=CWD” message shows that the user was in the `/var/mail` directory when they ran the `useradd` command (“CWD” is short for “current working directory”).

The “type=SYSCALL” message shows the program being executed, the user’s actual user ID that they logged in with (`auid=1000`– figure out the user name by looking in the `passwd` file for UID 1000) and the UID the command ran as (remember `uid=0` is root or admin level privileges). We also see here the “auth-files” keyword that the admin chose to use for changes to files like `/etc/passwd`.

```
ausearch -c <cmd> searches for a particular command name
ausearch -f <file> searches for messages related to a file (e.g. "ausearch -f
/etc/passwd")
ausearch -ua <uid> searches for messages related to a specific user ID
ausearch -k <keyword> lets you search for a specific keyword tag configured by the
system admin
```

If you are looking through audit logs in a non-standard directory path (like a mounted forensic image), use the “-if” option to specify the file or directory of files you wish to search rather than the default `/var/log/audit`.

Here is a quick listing of the more useful “type=...” messages found in audit logs:

- USER_AUTH, USER_LOGIN, USER_START ,USER_END, USER_LOGOUT – user interactive logins (SSH sessions also use CRYPTO_KEY_USER, CRYPTO_SESSION)
- USER_CMD, PROCTITLE, PATH, CWD, SYSCALL – process execution and user activity
- ADD_USER, ADD_GROUP – account admin activity
- AVC – SELinux messages
- TTY, USER_TTY – keystroke logs (if enabled)
- LOGIN, USER_ACCT, USER_START, USER_END, CRED_ACQ, CRED_DISP, CRED_REFR – related to scheduled task start/stop
- SYSTEM_BOOT, SYSTEM_RUNLEVEL, KERN_MODULE, NETFILTER_CFG
- DAEMON_START, SERVICE_START, CFG_CHANGE – system boot and startup messages

OTHER TOOLS

aureport

Generate summary reports for different event types
Get detailed breakdowns with **ausearch -a**

aulast

aulastlog

Produce output like **last** and **lastlog** using audit logs

In addition to **ausearch**, there is also the **aureport** command which produces a summary report of different sorts of activity. For example we could get a summary of all “type=SYSCALL” messages with “**aureport -s**” and then get more detail with **ausearch**:

```
# aureport -s -if /mnt/evidence/var/log/audit
```

```
Syscall Report
```

```
=====
# date time syscall pid comm auid event
=====
```

```
...
```

```
121. 02/20/2020 13:26:44 9428 1544 useradd 1000 342
```

```
...
```

```
# ausearch -if /mnt/evidence/var/log/audit -a 342
```

```
----
```

```
time->Thu Feb 20 13:26:44 2020
```

```
type=PROCTITLE msg=audit(1582223204.906:342) :...
```

aureport shows the audit ID number as the last field of each line item. “**ausearch -a**” lets you search by audit ID number.

If keystroke logging is enabled, you can dump the keystroke logs with `"aureport --tty"`.

`aulast` and `aulastlog` are like the `last` and `lastlog` commands we covered earlier. But instead of using the `wtmp` and `lastlog` files, `aulast` and `aulastlog` use audit log entries. This may be useful when attackers trash your `wtmp` file but forget about the audit logs.

LAB – MORE LOG ANALYSIS

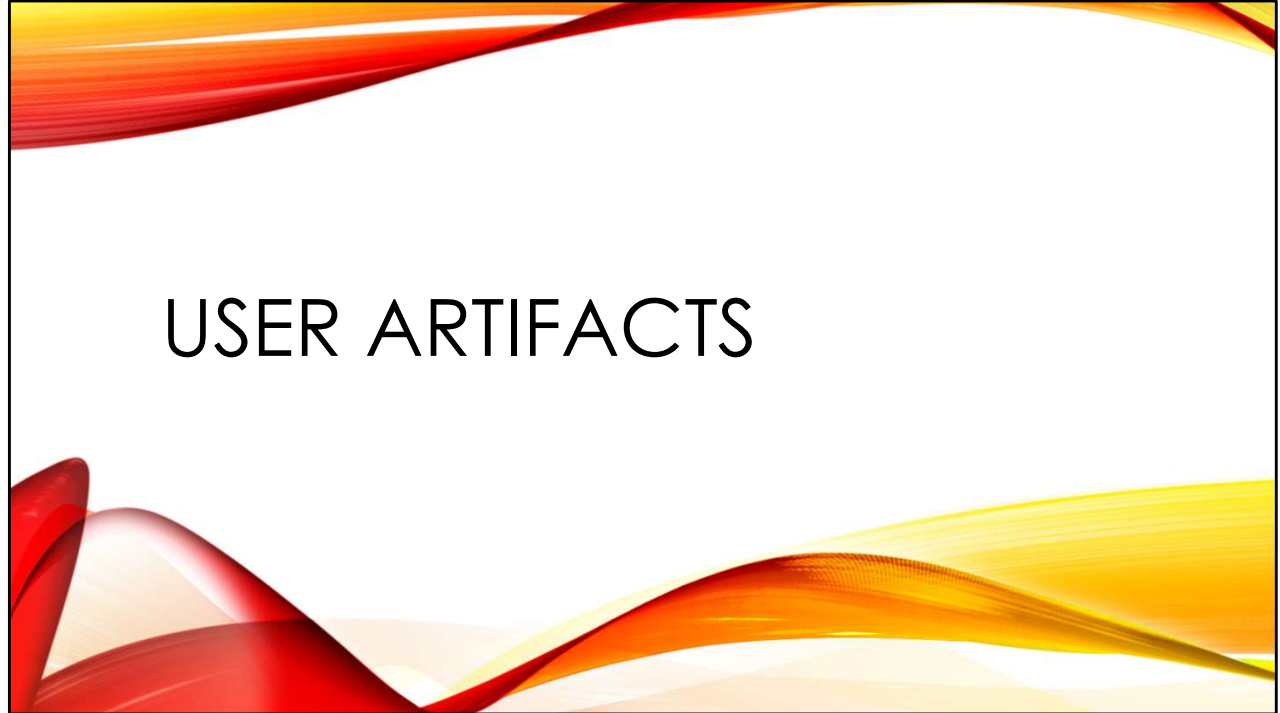
Adding more context

Adding web logs increases visibility into our intrusion!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



Tracking interactive user behavior is important during an investigation. There are multiple artifacts that can help.

COMMAND HISTORY

`$HOME/.bash_history`

Simple text file

Can easily be deleted or modified

Commands only written to file when shell exits

History in currently running shells is only in memory

bash_history may not be in chronological order

New file dropped on each shell exit

Search for previous versions in unallocated

The standard Linux command shell is bash and command history from the shell is saved in the file `.bash_history` in the user's home directory. The history file is just a simple text file and can be easily deleted. More worryingly, I can edit `bash_history` with a text editor and my modifications will be preserved even when the history gets updated with commands from later shells.

Note, however, that when a shell exits a brand new `bash_history` gets created and the old file rarely gets overwritten immediately. That means you can find plenty of older versions of `bash_history` floating around in unallocated (search for common command strings like `"cd /"` or `"rm -f"`). Use the `diff` command to compare the old `bash_history` you recovered against the current version and look at what changed.

Commands typed into the current shell are only saved when that shell exits. So the only way to get the command history from currently running shells is with the `linux_bash` plugin in Volatility. Also, commands may end up in `bash_history` out of chronological order. If I ran commands three days ago but never ended my bash session, those commands will go into `bash_history` *after* the commands I ran yesterday in a different shell that was closed down.

More information on `bash_history` forensics (and anti-forensics) at

http://www.deer-run.com/~hal/DontKnowJack-bash_history.pdf

A video of this presentation is <https://www.youtube.com/watch?v=ww1xqOV2RyE>

NO TIMESTAMPS!

When did a command happen?
Can't tell from `bash_history`!

Use other artifacts to pin down when commands occurred:

- Log entries
- File timestamps
- Process start times

The real difficulty is that there are no timestamps by default in `bash_history` (you can enable timestamps by setting the `HISTTIMEFORMAT` environment variable). Just looking at `bash_history`, you have no idea when the commands were executed. This is another huge point in favor of Volatility's `linux_bash` plugin because it always shows timestamps.

However, you can often associate commands in `bash_history` with other artifacts on the system. You might see a `useradd` command in `/root/.bash_history`. Audit log entries could tell you when that `useradd` command ran, as might the last modified time on `/etc/passwd` and `/etc/shadow`. If the user runs commands via `sudo`, you have those logs as well. You won't be able to pinpoint every command execution in `bash_history`, but if you can figure out several execution times, you can use these to "bracket in" chronologically the commands executed in between.

Once you have approximate time information, you can go back to `/var/log/wtmp` or your SSH or audit logs and figure out who was logged in at those times and from what remote IP address. This lets you attribute blocks of commands to particular user sessions.

SSH ARTIFACTS (1)

`$HOME/.ssh/authorized_keys`

INBOUND

Public keys that can be used to log into this system

Good place for attacker back doors

Key "comment" may give clues to source of key

SSH is the standard remote login protocol for Linux and Unix systems, and there are multiple SSH artifacts of interest.

The `authorized_keys` file holds public keys used for user authentication. One possible back-door for attackers is to add their own public key to `authorized_keys`—particularly the `authorized_keys` file for the root account. This gives them a legitimate login path into the system, in a place some admins wouldn't think to look.


Here's a sample `authorized_keys` entry:

```
ssh-rsa AAAAB3NzaC1y...qz3K0KvgmVbQ== hal@caribou
```

You have the key type (in this case an RSA key), the base64 encoded public key, and a comment. By default the comment contains the username of the user and the hostname of the machine where the key was made. In some cases, this could be useful attribution data. Note, however, the comment text could be easily changed with a simple text editor.

SSH ARTIFACTS (2)

`$HOME/.ssh/known_hosts`

OUTBOUND 

Public keys of system user has connected to from this host

Does not necessarily imply successful remote login

The `known_hosts` file tracks the public keys of systems a user has connected to from the local machine. Note that this does not necessarily mean a successful login to these remote systems— just that an SSH session was connected. Use the logs on the remote system (if available) to determine if the user successfully logged in.

By default the IP address of the remote system is clearly visible in each `known_hosts` entry. However, SSH does have a client option `HashKnownHosts` which hides the remote IP information using a one-way hash function. There is a brute-forcing tool that can be used to try and guess the hashed information:

https://github.com/halpomeranz/known_hosts_bruteforcer

SSH ARTIFACTS (3)

`$HOME/.ssh/config`

OUTBOUND

Can contain details about how to connect to other systems

`$HOME/.ssh/id_*`

OUTBOUND

Public/private key pairs for connecting to other systems
Correlate with **authorized_keys** entries from remote systems

A user's SSH `config` file can give details about usernames, keys, port numbers, and other settings necessary to connect to remote systems:

```
Host jumpbox
  Hostname jumpbox.sysiphus.com
  ServerAliveInterval 120
  Port 443
  IdentityFile /home/hal/.ssh/id_rsa-jumpbox
```

If I typed “ssh jumpbox” on my command line, I would be connecting to a machine called `jumpbox.sysiphus.com` via port 443/tcp rather than the default 22/tcp that SSH normally uses. Authentication would try to use the key in the `id_rsa-jumpbox` file rather than my default key (typically `$HOME/.ssh/id_rsa`).

Note that in addition to the encrypted private key in `id_*`, you will also find public keys in `id_*.pub` files. You should be able to match these public keys to `authorized_keys` entries on remote systems. Use the `known_hosts` entries to figure out which remote systems the user is connecting to.

FILE ACCESS/EDITING

\$HOME/.lesshst

Search terms
Shell escape commands

less is a "one screen at a time"
text viewing application

\$HOME/.viminfo

Recently accessed files (with position in file)
Command history
Search terms

vim is a standard
Linux text editor

`less` is a program like `more`, showing you one screenful of output at a time (there's an obscure joke here that "less is greater than more" because the `less` program has more functionality than `more`). The `less` program has its own history file that tracks search terms the user has typed in and commands the user ran via shell escapes from the `less` program. However, `lesshst` does *not* track which files the user is looking at.

`vim` is a standard Linux text editor. The `viminfo` file contains many useful artifacts, including recently edited file names along with the last position where the user was in the file. `viminfo` contains a history of search terms like `lesshst`, along with a history of `vim` commands the user type at the ":" prompt.

Linux text editors will also create backup copies of edited files. `vim` makes files with a `*.swp` extension, but other editors use a trailing tilde ("`passwd~`"). Running `diff` on the backup file vs the current version will quickly show changes made between the two versions.

DESKTOP ARTIFACTS

```
$HOME/.local/share/recently-used.xbel
```

Timestamped history of files opened with GUI applications

```
$HOME/.local/share/Trash/files
```

```
$HOME/.local/share/Trash/info
```

Files deleted with GUI are placed in “**files**”
“**info/***” files store original paths of deleted files

If users are using the Linux desktop, then they may be using the Linux file browser, called Nautilus or Nemo. The `recently-used.xbel` file is an XML formatted file that tracks files opened recently through the file browser, including the app used to open the file.

Files moved to the Trash folder via the GUI end up in `.../Trash/files`. The corresponding files under `.../Trash/info` say where the file originally came from.



WEB BROWSER ARTIFACTS

Firefox and Chromium are common browsers

Browser artifact formats don't change from Windows/Mac

Files under user home directories

Firefox: **`$HOME/.mozilla/firefox/*.default*`**

Chrome: **`$HOME/.config/chromium/Default`**

Desktop users may be using web browsers— Firefox and Chrome/Chromium are common on Linux. The good news here is that these web browsers create exactly the same history and cookie artifacts as the Windows and Mac versions. You could use any of the popular web browser forensic tools to extract and view this information.

LAB – USER (MIS)BEHAVIOR

Users' sordid histories

User artifacts can be a gold mine...or just an empty hole.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



EXT FILE SYSTEM FORENSICS

Some words about the internals of Linux EXT file systems and what remains once a file has been deleted– along with some strategies for recovering that data.

More detail can be found in the following articles:

<https://www.sans.org/blog/understanding-ext4-part-1-extents/>

<https://www.sans.org/blog/understanding-ext4-part-2-timestamps/>

<https://www.sans.org/blog/understanding-ext4-part-3-extent-trees/>

<https://www.sans.org/blog/understanding-ext4-part-4-demolition-derby/>

<https://www.sans.org/blog/understanding-ext4-part-5-large-extents/>

<https://www.sans.org/blog/understanding-ext4-part-6-directories/>

Additional detail on the older EXT3 file system can be found here:

<https://www.sans.org/blog/understanding-indirect-blocks-in-unix-file-systems/>

<https://www.fireeye.com/blog/threat-research/2011/01/ext3-file-recovery-indirect-blocks.html>



LET'S TALK ABOUT EXT

EXT4 is the modern incarnation of a very old file system

Much of what you will see is inherited from 4.2 BSD's FFS

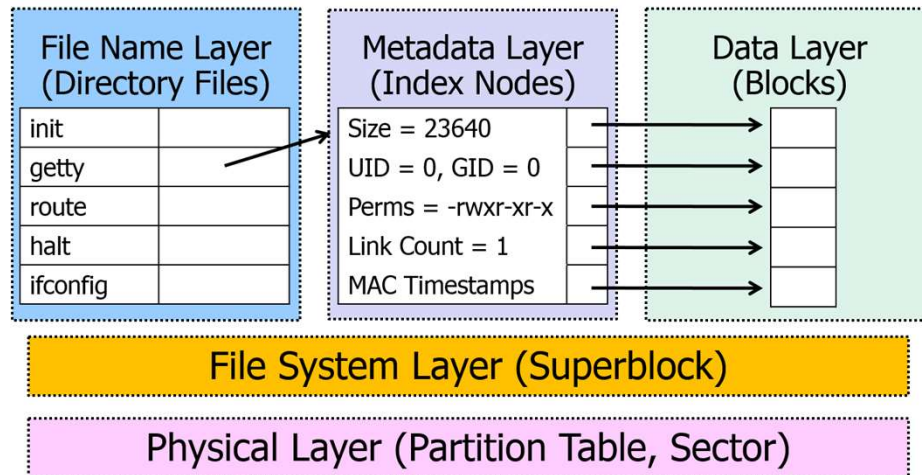
When the old and new worlds mix is when things get fun!

EXT4 is just the latest version of a very old Unix file system that traces its roots back to Marshall Kirk McKusick's "Fast File System" for Berkeley Unix. Similar file systems in other Unix flavors include FFS in BSD, UFS in Solaris and HP-UX, and so on. The tools we will be using in this section work with any of these common Unix file systems.

There are, of course, other file systems you may encounter on Linux, including ZFS, XFS, and Btrfs. Forensic support for these file systems is currently in the "limited to non-existent" state. Some detail about XFS forensics can be found on my blog at:

<https://righteousit.wordpress.com/tag/xfs/>

EXT FILE SYSTEM LAYERS



You're probably familiar with the "seven-layer" OSI model for describing network communications. File systems can be conceptualized as a "five-layer" model:

- **Physical Layer**: The physical drive or device and the partitions on it. Partition geometry is described by a *partition table* at the beginning of the disk—sometimes referred to as a *Volume Table of Contents (VTOC)* or *disk label*. *Sectors* are the smallest unit of storage addressable by the disk controller.
- **File System Layer**: Contains all the configuration and management data associated with the file systems in each partition on the disk. For Unix file systems, the primary structure of interest at this layer is an object called a *superblock*.
- **The File Name Layer (AKA Human Interface Layer)** is responsible for mapping human-readable file names to metadata addresses. In Unix file systems, this is accomplished with special *directory files* that map file names to *index node (inode)* numbers in the layer below.
- **Metadata Layer**: Contains all of the data structures that are responsible for the definition and delineation of files. In Unix file systems, we use objects called *index nodes (inodes for short)* that store metadata about files and pointers to the disk blocks that make up the contents of the file
- **Data Layer**: Contains the actual data units of disk storage—commonly referred to as *blocks* in Unix file systems (Windows file systems use the term *clusters* instead of *blocks*).

PHYSICAL LAYER: DISK PARTITIONS

A disk can be segmented into *partitions*

Partition table at beginning of the disk provides a map

Each partition is treated as an independent device in Linux

A partition usually contains a file system or swap

The physical layer consists of the physical disk device and the structures that define it.

A disk drive (with a file system) must contain at least one partition, though it may be segmented into many. The front of each disk contains a *partition table* or *Volume Table of Contents* (VToC). BSD systems may also have their own BSD-specific *disk label* in addition to the basic partitioning information for the drive.

Linux systems often use the old DOS *Master Boot Record* (MBR) style partitions with four “primary” partitions and chained “extended” (logical) partitions as necessary. GPT (GUID Partition Tables) is a newer disk partitioning scheme designed to overcome many of the limitations of traditional MBR-style partition tables, and may be found on some Linux systems.

Even though multiple partitions may exist on the same disk, the Unix operating system treats them as independent devices and performs file I/O via individual entries in the `/dev` directory—e.g., `/dev/sda1`, `/dev/sda2`, and so on.

Some partitions contain a file system like EXT4, though as we’ve seen more complicated RAID and LVM configurations are common too. Sometimes a file system is not formatted—for example, Unix swap partitions will typically use “raw” partitions, and some databases use raw partitions to try to improve performance.

FILE SYSTEM LAYER: SUPERBLOCK

The File System Layer contains data that describes the file system within a partition

Unix uses a *superblock* that contains the following data:

- FS type/size, block size, number of blocks/inodes, etc.
- Modification time, last mounted on, clean/dirty status
- Pointer to inode for file system journal (EXT3 and above)

When a file system is created in a partition, a data structure is created at the beginning of the partition to define the attributes of the file system that resides there. For Unix file systems, this data structure is called a *superblock*.

The superblock is a 512 byte data structure that can be found offset 1024 bytes from the front of the file system (the offset was designed to help protect the file system from somebody accidentally overwriting the front of the drive). The superblock contains basic file system information including items like the file system type, block size, the number of blocks and inodes in the file system, the number of unallocated blocks and inodes, and so on. It also contains information about the usage of the file system, like when it was last mounted, where it was mounted, whether it was unmounted cleanly, and so on. The superblock is always replicated in multiple locations throughout the file system, to provide fault tolerance against disk failure in the first superblock.

For EXT3 and later file systems, the superblock contains a pointer to the inode of the file system journal (though this is always inode number 8 on Linux file systems). The superblock *does not* contain a pointer to the inode of the root of the file system, because—by convention—inode 2 is reserved for the root directory (in older versions of the Unix file system, the "file" at inode number 1 was used to store bad block addresses).

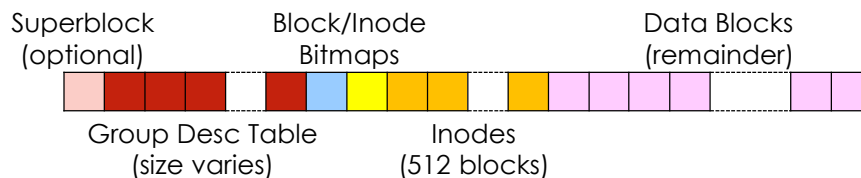
BLOCK GROUPS

Blocks are organized into *Block Groups* of 32K blocks

Each block group contains inodes and data blocks

Block and inode allocation bitmaps each occupy 1 block

May also contain backup superblock, etc



In addition to the superblock, EXT file system metadata also includes the *block group descriptor table*. Data blocks in the file system are grouped into *block groups* of 32K blocks. Within each block group, some blocks are set aside for storing *inodes* which hold file metadata (timestamps, ownerships, permissions, etc) and some blocks for data (file content).

Typically there are 8K inodes in each 32K block group (one inode per four data blocks is the usual ratio, though this can be tuned when the file system is created). The default block size in EXT4 is 4K, and EXT4 inodes are 256 bytes in size, so 512 blocks of the block group are dedicated to holding inodes.

In addition, one block at the front of the block group is the block bitmap that tracks the allocation status of each block in the group. A single 4K block has 32K bits for tracking individual blocks, which is why the default block group size is 32K blocks. There is also another block dedicated to tracking the allocated status of each inode.

Some block groups may also contain a backup of the superblock and block descriptor table. These would consume additional blocks at the front of the block group, prior to the block and inode allocation bitmaps.



DATA LAYER: BLOCKS

Data Layer is for storing files' contents

The basic storage unit is a *block*

Blocks are composed of sectors (usually 8 in EXT)

This is the smallest unit of file I/O in the file system

For efficiency, the blocks that make up a file are allocated consecutively when possible

The data layer is where the binary information is actually stored on disk. The smallest storage unit addressable by the disk device is a *sector* that is usually 512 bytes. However, to improve I/O performance, EXT file systems will normally perform reads/writes in 4K chunks called *blocks*.

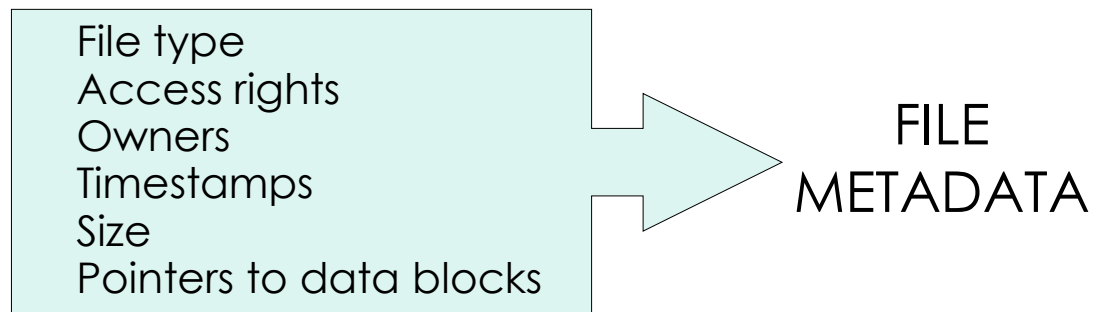
On modern Linux file systems, the standard 4K block size is the minimum amount of data that will be allocated to any file. If the file is smaller than 4K, the remainder of the block is unused or "slack" space. Slack space is null filled on Linux file systems, so do not expect to recover data from Linux file slack like you might in Windows. However, long runs of nulls are often a good way of figuring out when you've reached the end of a file.

When writing a large file that spans multiple blocks, the file system will try to allocate consecutive blocks where possible. This will increase the read efficiency because the file system can "read ahead" in large swaths. But this tendency also turns out to be useful when we're trying to recover deleted data. If you can locate a suspicious string in the middle of a "deleted" block of data, you may be able to recover the entire deleted file by capturing the blocks immediately before and after the "interesting" block. Recall, however, that data blocks are organized into block groups of 32K blocks and any file larger than this will be forced to "fragment" into multiple chunks.

METADATA LAYER: INODES

Metadata Layer stores "non-content" data about files

Uses structures are called *inodes*—every file has one



The contents of the file are important, but so are the other non-content parameters associated with the file— timestamps, file ownership and permissions, file size and type, etc. Unix file systems use *index nodes*— commonly shortened to *inodes*— to store this data. Inodes store everything about the file that you are used to seeing in the output of "`ls -l`" *except* for the file name. An inode also has pointers to the data blocks that make up the contents of the file.

As we saw earlier, inodes are stored in data blocks in each block group. Each inode has an address—they are simply numbered sequentially. You can see the inode number associated with each file using "`ls -li`".



FILE NAME LAYER: DIRECTORIES

Partition (major/minor device number) and inode number are how the kernel tracks files

Humans don't like accessing files via sequences of numbers

Directory files associate file names with inode numbers

The Unix operating system tracks files using the inode number and the device numbers associated with the disk partition that holds the file. But human beings don't find a series of numbers convenient for naming files, so some interface layer between humans and machines is necessary.

The Human Interface (or File Name) Layer contains special file system objects whose purpose is to associate human-readable file names with the inode numbers used by the OS. The "special objects" are what we call directories...

DIRECTORIES

Ext4 dirs are unsorted lists of records:

Inode number (4 bytes)
4-byte aligned entry length (2 bytes)
File name length (1 byte)
File type (1 byte)
File name

Byte Offset in Directory	Inode Number	File Name
0	84	.
12	6	..
24	1854	fdisk
40	1232	fsck
52	87	halt
64	1789	lsmmod
80	1523	mkfs
92	74	mount
108	1466	reboot
124	132	umount
140	90	syslogd
156	1656	ifconfig

Directories are simply special files that associate file names with inodes. In the traditional Unix file systems, a directory "file" is just a sequential list of records that store file names along with their corresponding inode. When you list a directory, you are basically just dumping the contents of the directory "file". Note that the directory records also contain a byte for the file type so that commands like `"ls -F"` don't have to read the inode for each file to produce output.

Directories give the file system its hierarchical structure. Consider what happens in the operating system when you try to access a file like `/home/hal/.profile`:

- Remember that inode 2 is reserved for the root of the file system, so the file system driver begins by opening this "file".
- The OS reads the contents of the root directory "file" pointed to by this inode until it finds the entry for "home" and the associated inode with this entry.
- The OS then opens the inode from the "home" directory entry—this is another directory "file" and the OS scans through the contents until it finds the "hal" entry and its inode.
- Now we have the inode for `/home/hal`—yet another directory, so the OS has to scan through the directory to find the entry for `".profile"`.
- Finally, the OS has the inode for `/home/hal/.profile` so it can open this file and read its contents.

DELETING A FILE

Directory entry for deleted file *unchanged*

Previous directory entry “grows” to consume space

Result: See the file name and inode of deleted files!

You may be wondering why the directory records track both the file name length and the total record length. Firstly, directory records must be 4-byte aligned—so the entry for the “.” link, which could fit into 9 bytes, is padded out to 12 bytes with 3 bytes of slack at the end of the record.

But, more interestingly, when a file is deleted in Unix file systems, its directory entry is not changed. All that happens is that the length of the previous record is extended to consume the space the entry of the deleted file. The record for the deleted file is still visible in the “slack” space of the previous directory entry (at least until some other file is created in the directory that consumes this slack space). So if the length of an entry is at least 12 bytes longer than would otherwise be necessary, then you will find the entries for one or more deleted files in the excess space— inode, file type, and file name.



THE BAD NEWS

Extent data is zeroed when files are deleted in Ext4

Knowing the inode of the deleted file doesn't help!

Or does it..?

Unfortunately, knowing the inode of the deleted file is not as helpful as it sounds. Starting with EXT3, the block pointers in the inode are zeroed when the file is deleted. This was a deliberate design decision for privacy—making it more difficult to recover the file content of deleted files. So while we can see the inode number of the deleted file in the directory entry found in the slack of the directory file, we can't use this inode number to directly recover the file content.

One other note about inode changes when a file is deleted— the mtime and ctime fields in the inode are set to the time of file deletion. However, atime and btime are untouched. Note that EXT3 and later also have a 32-bit deletion time field (seconds resolution only) in the inode, which is also set to the time of file deletion.



ALLOCATION STRATEGY

New directories are created in the least used block group

New files are added to same block group as directory

It turns out that classic Unix file systems try to keep the data blocks associated with a file in the same block group as the inode for the file. Historically, each new directory would be assigned to the least used block group. The files within that directory would be created in the same block group with their parent. If the files in a single directory consumed all the blocks in the block group, new files would just be created in the next physical block group on the drive.



DELETED DATA

1. Use directory entry to determine inode of deleted file
2. Determine block group number from inode number
3. Search block group unallocated for deleted data
4. Profit?

So if we know the inode number of the deleted file by looking at the deleted directory entry, we have a clue about where the contents of the deleted file are likely to be on disk. This allows us to significantly narrow the amount of blocks we need to search through to find the deleted file. It's very likely that the file content is in the same 32K block group where the inode is located, or perhaps in the next physical block group.

LAB – RECOVER DELETED FILE

Gone but not forgotten!

Let's use what we've learned about EXT file systems to try and recover some interesting data from our hacked web server image. We'll learn some new tools along the way!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



XFS FILE SYSTEM FORENSICS

XFS is another popular file system for Linux systems. Unfortunately, our forensic tools are limited when it comes to this file system.

For a much deeper dive into XFS than I am able to do here, see my series of blog posts at <https://righteousit.wordpress.com/tag/xfs/>



ABOUT XFS

High-performance file system, originally created by SGI

Common in NAS appliances

Default file system for Red Hat distributions

Forensic tool support?

X-Ways?

Sleuthkit branch?

The XFS file system was originally created by Silicon Graphics Inc for their IRIX operating system. When SGI migrated into the Linux space, they brought XFS along with them.

These days, XFS is a popular choice for a high-performance Linux file system. Many lower-end Network Attached Storage (NAS) devices are simply Linux systems with disk arrays attached running Linux software RAID and XFS. XFS also got a big popularity boost when Red Hat announced that XFS would be the default file system starting with RHEL 7.

Unfortunately, there is only very limited forensic tool support for XFS. X-Ways does support XFS and there is a development branch of the Sleuthkit that is working towards XFS support (<https://github.com/sleuthkit/sleuthkit/pull/1476>). However, I believe that both products need significantly more testing when it comes to XFS file systems. Be careful to verify your findings when working with either tool.



KEY FEATURES

- Journaling file system
- 64-bit addressing
- Inodes allocated as needed
- Extent-based file allocation
- MACB timestamps with nanosecond resolution
- All file system data structures are big-endian

XFS is a modern, journaling file system. It uses 64-bit addressing and so has the potential to create massive volumes and files— theoretically into the exabyte range.

Like EXT, the default block size for XFS is 4K, but XFS inodes are 512 bytes, making them much larger than even EXT4 inodes. This large inode size gives XFS the potential to have resident data in the inode similar to NTFS. The current XFS implementation will keep small directories resident in the inode but there is no support for resident data files.

Also XFS does not pre-allocate inodes like traditional Unix file systems. When more inodes are needed, XFS will simply grab a group of four data blocks (16K) and designate them for inode storage. This tends to keep file metadata close to the file data and means less overhead in the file system for inode storage. Inode addressing is based on physical offset from the start of the file system and so inode numbering is not sequential as it is in traditional Unix file systems.

Like EXT4, XFS supports MACB timestamps at nanosecond resolution. In the current production version XFS uses 32-bit seconds in Unix epoch time and 32-bit fractional seconds, which means XFS has the traditional Unix 2038 date rollover problem. The current development work for XFS (released in kernel 5.10) includes a feature called "bigtime" that changes the internal file system date format to 64-bit nanosecond counters with a December 1901 epoch, which pushes the maximum date out to year 2486.

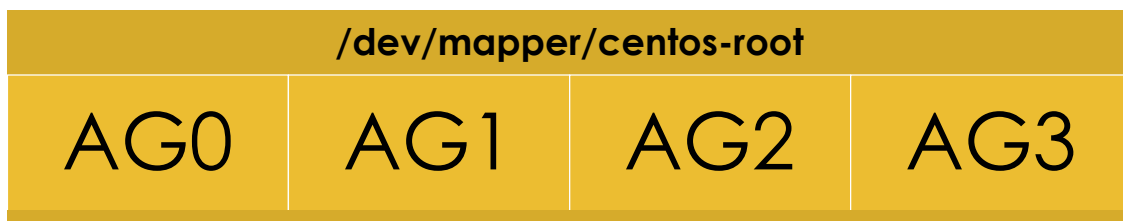
All XFS file system data structures and addresses are stored in big-endian format regardless of the processor architecture. XFS was originally developed by SGI for their MIPS RISC processors, which were big-endian CPUs. Frankly, this feature isn't going to matter much to you unless you spend a lot of time looking at XFS data in a hex editor (like I do, and I love it!).

WHAT'S DIFFERENT ABOUT XFS?

Each file system made up of several *Allocation Groups (AGs)*

Each AG can be written independently

Allows parallel writes for faster throughput



What's really different about XFS compared to other file systems is that each XFS file system is broken up into multiple *allocation groups* or *AGs*. XFS defaults to four AGs per file system, but you can tune this when you are creating the file system.

Each AG is effectively its own file system that can be written to independently of the other AGs. This allows multiple threads to be writing into different AGs without interfering with each other or needing to take out locks. This gives XFS very fast throughput on multi-core CPU architectures.

The primary superblock for the file system is located at the beginning of AG 0 and duplicate superblocks are kept at the beginning of the other AGs for redundancy.

BLOCK AND INODE ADDRESSING

Addresses are packed structures

Upper bits hold the AG number

Lower bits are the AG relative block offset

Field lengths are *variable*, based on AG size in file system

AG Number

Block address from start of AG

XFS block and inode addresses are more complicated because they need to specify both the AG that the file belongs to and the block offset within that AG. Standard XFS addresses are a packed 64-bit value with the block or inode number in the lower bits and the AG number in the upper bits.

The tricky part is that the block portion of the address is *variable length* based on the size of the AG.

```
[lab@LAB CentOS-XFS]$ xfs_db -r centos-root.raw
xfs_db> sb 0
xfs_db> print dblocks
dblocks = 2621440
xfs_db> print agcount
agcount = 4
xfs_db> print agblocks
agblocks = 655360
xfs_db> print agblklog
agblklog = 20
```

Here I am using the low-level `xfs_db` program to interrogate the primary superblock ("sb 0") of an XFS file system image. This is small (roughly 10GB) file system containing 2621440 total 4K blocks. The blocks are divided evenly between 4 AGs, so each AG contains 655360 blocks. The file system needs 20 bits to address 655260 blocks ($\text{ceil}(\log_2(655360)) = 20$). So in this file system, the lower 20 bits of each 64-bit address will be the block number relative to the beginning of the AG and the upper 44 bits will hold the AG number.

With standard 4K data blocks and 512 byte inodes, each data block can hold 8 inodes. That means inode addresses need an extra three bits to track all possible inode numbers. So in this particular file system, the 64-bit inode addresses will use the lower 23 bits for the inode number relative to the start of the AG.

Even in a much larger file system, it's rare for the inode or block addresses to require even 32 bits of the 64-bit address for the relative address plus AG number combo. That means the upper 32 bits of the address space is often wasted. We will be exploiting this feature at the end of this material when we start talking about recovering deleted data.

LOOKING FOR TREASURE

Learn about XFS tools and addressing with a case study
Located a string of interest in a file system image
What file is this string found in?

Start by converting byte offset into sector offset

```
[lab@LAB ~]$ cd /images/All-Images/CentOS-XFS/  
[lab@LAB CentOS-XFS]$ strings -a -t d centos-root.raw | gzip >strings.asc.gz  
[lab@LAB CentOS-XFS]$ zgrep -Fi treasure strings.asc.gz  
[... snip ...]  
9010062352                                Unit 1/2/3, 20/F, New Treasure Center  
[... snip ...]  
[lab@LAB CentOS-XFS]$ expr 9010062352 / 512  
17597778
```

XFS addressing can be confusing. Also, with limited forensic tool support for XFS you will often need low-level XFS file system tools to verify your results.

We will work through an example together to review XFS addressing concepts and introduce the `xfs_db` tool for examining XFS file systems. Suppose we found a string of interest at a particular byte offset in a file system. Using `xfs_db`, can we sufficiently reverse-engineer the file system to get from this byte offset to the actual file name that contains our string of interest? Of course we can!

As you can see on the slide, I have created a sample XFS file system image to experiment with. Using the "strings" command, I extract ASCII strings and their byte offsets ("-t d") from the image. Then we go looking for "treasure" in the file system strings. Obviously this is a common word and we will get lots of hits, but one of the strings seems to be a physical address and that might be fun to search for.

The string starts at byte offset you see on the slide. But to get started with `xfs_db` we are going to need a sector offset. We calculate this by dividing the byte offset by the sector size in bytes.

XFS_DB CONVERTS ADDRESSES

```
[lab@LAB CentOS-XFS]$ xfs_db -r centos-root.raw
xfs_db> convert daddr 17597778 fsblock
0x3390aa (3379370)
xfs_db> convert fsblock 3379370 agno
0x3 (3)
xfs_db> convert fsblock 3379370 agblock
0x390aa (233642)
```

daddr	Sector offset
fsblock	Packed AG+block num
agno	AG number only
agblock	AG relative block num

XFS addressing is complicated, but fortunately `xfs_db` allows us to convert between different addressing formats. However, because the relative block and inode address portion is variable length depending on the size of the file system, you must open the file system with `xfs_db` before you can begin converting addresses.

Here we are opening our file system image with `xfs_db` in read-only mode ("-r"). You can use this same option to attach `xfs_db` to a running file system on a live machine.

`xfs_db` refers to the absolute sector offset as the "daddr" (direct address?) value. Having calculated this daddr value from the byte offset of the string, we can use `xfs_db` to convert the daddr value into the corresponding "fsblock" address (XFS "file system block" number). "fsblock" addresses are the standard packed address format with the AG number in the upper bits and the relative block number in the lower bits.

`xfs_db` also allows us to deconstruct fsblock addresses into the AG number ("agno") and relative block address ("agblock"). You can even convert daddr addresses directly into agno and agblock addresses.

XFS_DB CONTENT PREVIEW

```
xfs_db> daddr 17597778
xfs_db> type text
xfs_db> print
000:  6c 65 63 74 72 6f 6e 69 63 73 20 4c 74 64 2e 0a  lectronics.Ltd..
010:  09 09 09 09 55 6e 69 74 20 31 2f 32 2f 33 2c 20  ....Unit.1.2.3..
020:  32 30 2f 46 2c 20 4e 65 77 20 54 72 65 61 73 75  20.F..New.Treasu
030:  72 65 20 43 65 6e 74 65 72 0a 0a 09 09 09 09 48  re.Center.....H
040:  4b 0a 0a 30 30 2d 31 41 2d 35 39 20 20 20 28 68  K..00.1A.59....h
[... snip ...]
```

xfs_db allows us to examine low-level XFS structures. For data structures like superblocks and inodes, xfs_db will automatically parse out the fields of each data structure and present them in a human-readable format. Try typing the command "sb 0" to select the primary superblock and then "print" to print out all of the fields.

You can also use xfs_db to preview raw sectors and blocks. Here we are selecting our raw sector object as the item to display, but we could have also used the fsblock address we calculated on the previous slide and dump the entire 4K block. "type text" means to display the output in the standard hex dump format that you see on the slide.

Currently xfs_db does not support simply dumping the raw data. However, we can use the "dd" command for this if necessary:

```
[lab@LAB CentOS-XFS]$ dd if=centos-root.raw bs=512 skip=17597778
count=1
lectronics Ltd.
```

Unit 1/2/3, 20/F, New Treasure Center

HK

Dumping the raw block with "dd" requires doing some arithmetic based on the agno and agblock addresses and the number of blocks per AG (the "agblocks" value from the superblock). The absolute block position in the file system can be expressed as "agno * agblocks + agblock".

Happily, we can use the shell to do this arithmetic for us:

```
[lab@LAB CentOS-XFS]$ dd if=centos-root.raw bs=4096
                        skip=$((3*655360 + 233642)) count=1
[... snip ...]
001A54      (base 16)      Hip Shing Electronics Ltd.
                        Unit 1/2/3, 20/F, New Treasure
Center
                        HK
[... snip ...]
```

BLOCKGET/BLOCKUSE FTW!

```
xfs_db> blockget -n -s
xfs_db> fsblock 3379370
xfs_db> blockuse -n
block 3379370 (3/233642) type data inode 25629955 usr/share/hwdata/oui.txt
```

```
[root@LAB CentOS-XFS]# mkdir -p /mnt/xfs
[root@LAB CentOS-XFS]# mount -o ro,noexec,loop centos-root.raw /mnt/xfs
[root@LAB CentOS-XFS]# grep -F -C2 'New Treasure' /mnt/xfs/usr/share/hwdata/oui.txt
00-1A-54      (hex)          Hip Shing Electronics Ltd.
001A54       (base 16)       Hip Shing Electronics Ltd.
                                Unit 1/2/3, 20/F, New Treasure Center

                                HK
```

xfs_db also allows us to track the allocation status of blocks and figure out which inode/file a given block is associated with.

Start by using "blockget" to create a mapping between blocks and inode numbers. The "-n" option means to also track the file names associated with each inode. "-s" means be silent unless there are critical errors, otherwise you get a ton of noisy debugging output.

Then simply specify an fsblock address and then run "blockuse -n" to show the inode and file name ("-n") that block is associated with. Too easy!

To validate our finding, we can mount the XFS file system image and grep for our string of interest in the file that "blockuse" gave us. Woot! There it is!

WHAT ABOUT DELETION?

Directory

- Entry marked as free space
- Inode field partially overwritten but still readable

Inode

- ctime updated to deletion time
- File size, num extents zeroed
- Extent data not overwritten

Our last case study worked easily because the string we were looking for was in a file that was allocated and part of the file system. But what about finding deleted data?

Similar to EXT file systems, when a file is deleted its entry in the directory "file" is not overwritten but instead simply marked as free space. In XFS that means overwriting the first 16 bits of the inode address with 0xFFFF to indicate free space, followed by a 16-bit value indicating the length of the free section. So the upper 32-bits of the inode address get clobbered. But fortunately, the inode addresses for even fairly large file systems (up to 512GB or so) fit into 32 bits and we can just read the inode out of the remaining portion of the address. For larger file systems, you can lose the bits that are the AG number and simply check the relative block address in all AGs.

When the file's inode is deallocated, information about the file size and number of extents is zeroed out. But the actual extent data structures in the inode are not overwritten. So if we can inspect the raw inode, we can discover the blocks where the file used to exist on disk!

Sounds like a plan! Let's test our theory with a fun little lab exercise...

LAB – XFS ARCHEOLOGY

Find the (deleted) flag!

Recover deleted data from an XFS file system using only primitive tools? Sounds like fun to me!

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.

LAB – HONEYPOT CAPSTONE

Choose your own adventure!

You've learned a lot! Now put those skills to use doing further investigation with Tyler Hudak's compromised honeypot data– with the disk image now in addition to the memory and UAC data we've used in previous labs.

You'll find the exercises as HTML files under `/home/lab` in your Virtual machine:

1. Launch the Firefox web browser
2. Use `Ctrl-O` to open a file
3. Navigate to `/home/lab/Exercises` and open `index.html`
4. Click on the link to go to the appropriate Exercise

Exercise HTML files are also in the `Exercises` directory on the course USB. Some people prefer to open the Exercise in a browser on their host operating system rather than in the virtual machine.



THANK YOU!

Any final questions?
Thanks for listening!

hrpomeranz@gmail.com
@hal_pomeranz@infosec.exchange



Attribution-ShareAlike
CC BY-SA

I hope you learned a lot from this material and had some fun along the way.

If you have questions or feedback in the future, please don't hesitate to contact me:

Hal Pomeranz
hrpomeranz@gmail.com
@hal_pomeranz@infosec.exchange

Download updates from <https://archive.org/details/HalLinuxForensics>